

Rust par l'exemple

Par Rust Core Team  - Anthony Defranceschi (traducteur)

Date de publication : 23 avril 2018

Rust est un langage système moderne mettant l'accent sur la sécurité, la vélocité et la concurrence. Il y parvient en gérant la mémoire sans utiliser de ramasse-miettes(garbage collector).

RBE est une collection d'exemples parfaitement exécutables qui illustre les différents concepts du langage Rust et ses bibliothèques standards.

Pour en savoir plus à propos des exemples présentés, n'oubliez pas **d'installer Rust sur votre machine** et de visiter **la documentation officielle**. Pour les plus curieux, vous pouvez également **consulter le code source du site**.

Ce document est une traduction du livre ***https://doc.rust-lang.org/rust-by-example/index.html****.*

1 - Hello World

Voici le code source d'un traditionnel « Hello World ».

```
// Ceci est un commentaire, et sera ignoré par le compilateur.

// Ceci est la fonction principale
fn main() {
// Toutes les déclarations se trouvant dans le corps de la fonction
// seront exécutées lorsque le binaire est exécuté.
// Afficher du texte dans la console.
    println!("Hello World!");
}
```

println! est une macro qui affiche du texte sur la console.

Un binaire peut être généré en utilisant le compilateur Rust : rustc.

```
$ rustc hello.rs
```

rustc va produire un binaire nommé « hello » qui pourra être exécuté :

```
$ ./hello
Hello World!
```

Activité

Cliquez sur le bouton « Run » en début de section pour visualiser le résultat présenté. Ensuite, ajoutez une nouvelle ligne qui permettra de visualiser le résultat ci-dessous :

```
Hello World!
I'm a Rustacean!
```

1-1 - Les commentaires

N'importe quel programme a besoin de commentaires, c'est pour cela que Rust supporte différentes syntaxes :

Les commentaires basiques ignorés par le compilateur :

- // Les commentaires mono-lignes. ;
- /* Les blocs de commentaires régis par leurs délimiteurs. */.

Les commentaires dédiés à la documentation qui seront convertis au format HTML :

- /// Génère de la documentation pour ce qui suit ce commentaire. ;
- //! Génère la documentation pour un conteneur (e.g. un module) ;
- /*! Permet de rédiger un bloc entier de documentation.*/.

1-2 - Affichage formaté

L'affichage est pris en charge par une série de macros déclarées dans le module std::fmt qui inclut :

- format! : Construit la chaîne de caractères du texte à afficher ;
- print! : Fait exactement la même chose que format!, mais le texte est affiché dans la console ;
- println! : Fait exactement la même chose que print!, mais un retour à la ligne est ajouté.

Toutes formatent le texte de la même manière.

Note : la validité du formatage (i.e. Si la chaîne de caractères que vous soumettez peut être formatée comme vous le désirez) est vérifiée au moment de la compilation.

```
fn main(){
    // En général, le marqueur '{}' sera automatiquement remplacé par
    // n'importe quel argument. Il sera transformé en chaîne de caractères.
    println!("{} jours", 31);

    // Sans suffixe, 31 est de type i32. Vous pouvez changer le type de 31 avec
    // un suffixe. (e.g. 31i64)

    // Différents modèles peuvent être utilisés.
     // Les marqueurs de position peuvent être utilisés.
     println!("{0}, voici {1}. {1}, voici {0}", "Alice", "Bob");

     // Les marqueurs peuvent également être
     // nommés
     println!("{sujet} {verbe} {objet}",
     objet="le chien paresseux",
     sujet="Rapide, le renard",
     verbe="saute par-dessus");

     // Un formatage spécial peut être spécifié après un ':'.
     println!("{} personne sur {:b} sait lire le binaire, l'autre moité non.", 1, 2);

     // Vous pouvez aligner vers la droite votre texte en spécifiant
     // la largeur (en espace) entre le côté gauche de la console
     // et votre chaîne. Cet exemple affichera: "     1", un "1" après 5 espaces.

     println!("{number:>width$}", number=1, width=6);

     // Vous pouvez également remplacer les white spaces par des '0'.
     // Affiche: "000001"

     println!("{number:>0width$}", number=1, width=6);

     // Le nombre d'arguments utilisé est vérifié par le compilateur.
     // println!("Mon nom est {0}, {1} {0}", "Bond");
     // FIXME ^ Ajoutez l'argument manquant: "James".

     // On créé une structure nommé 'Structure' contenant un entier de type 'i32'.
     #[allow(dead_code)]
     struct Structure(i32);

     // Cependant, les types complexes tels que les structures demandent
     // une gestion de l'affichage plus complexe. Cela ne fonctionnera pas.
     // println!("Cette structure '{}' ne sera pas affichée...", Structure(3));
     // FIXME ^ Commentez/Décommentez cette ligne pour voir le message d'erreur.
}
```

std::fmt contient plusieurs traits qui structurent l'affichage du texte. Les deux plus « importants » sont listés ci-dessous :

1. **fmt::Debug** : Utilise le marqueur {:?}. Applique un formatage dédié au débogage.
2. **fmt::Display** : Utilise le marqueur {}. Formate le texte de manière plus élégante, plus « user friendly ».

Dans cet exemple, fmt::Display était utilisé parce que la bibliothèque standard fournit les implémentations pour ces types. Pour afficher du texte à partir de types complexes/personnalisés, d'autres étapes sont requises.

Activité

Réglez les deux problèmes dans le code ci-dessus (cf. FIXME) pour qu'il s'exécute sans erreurs.

Ajoutez une macro println! qui affiche : « Pi est, à peu près, égal à 3,142 » en contrôlant le nombre affiché de chiffres après la virgule. Dans le cadre de l'exercice, vous utiliserez let pi = 3.141592 comme estimation de Pi (**Note** : vous

pourriez avoir besoin de consulter la documentation du module **std::fmt** pour configurer le nombre de décimaux à afficher).

Voir aussi

std::fmt, **les macros**, **les structures**, **les traits**.

1-2-1 - Debug

Tous les types qui utilisent le formatage des traits du module std::fmt doivent en posséder une implémentation pour être affichés.

Les implémentations ne sont fournies automatiquement que pour les types supportés par la bibliothèque standard. Les autres devront l'implémenter « manuellement ».

Pour le trait fmt::Debug, rien de plus simple. Tous les types peuvent hériter de son implémentation (i.e. la créer automatiquement, sans intervention de votre part). Ce n'est, en revanche, pas le cas pour le second trait : fmt::Display.

```
// Cette structure ne peut être affichée par `fmt::Debug`,
// ni par `fmt::Display`.
struct UnPrintable(i32);

// L'attribut `derive` créé automatiquement l'implémentation requise
// pour permettre à cette structure d'être affichée avec `fmt::Debug`.
#[derive(Debug)]
struct DebugPrintable(i32);
```

Également, tous les types de la bibliothèque standard peuvent être automatiquement affichés avec le marqueur {:?} :

```
// On fait hériter l'implémentation de `fmt::Debug` pour `Structure`.
// `Structure` est une structure qui contient un simple entier de type `i32`.
#[derive(Debug)]
struct Structure(i32);

// On créé une structure nommée `Deep`, que l'on rend également affichable,
// contenant un champ de type `Structure`,
#[derive(Debug)]
struct Deep(Structure);

fn main() {
    // L'affichage avec le marqueur `{:?}` est similaire à `{}`,
    // pour des types standards comme les entiers et les chaînes de caractères.
    println!("{:?} mois dans une année.", 12);
    println!("{1:?} {0:?} est le nom de {actor:?}.",
             "Slater",
             "Christian",
             actor="l'acteur");

    // `Structure` peut être affichée !
    println!("{:?} peut désormais être affichée!", Structure(3));

    // Le problème avec `derive` est que vous n'avez aucun contrôle quant au résultat
    // affiché. Comment faire si je souhaite seulement afficher `7` ?
    println!("{:?} peut désormais être affichée!", Deep(Structure(7)));
}
```

Finalement, fmt::Debug permet de rendre un type personnalisé affichable en sacrifiant quelque peu « l'élégance » du résultat. Pour soigner cela, il faudra implémenter soit-même les services du traits fmt::Display.

Voir aussi

Les attributs, **derive**, **std::fmt**, **les structures**.

1-2-2 - Display

fmt::Debug propose un formatage rudimentaire, et il peut être de bon ton de soigner ce que nous affichons. Pour ce faire, il faudra implémenter fmt::Display (qui utilise le marqueur {}).

Voici un exemple d'implémentation du trait :

```
// On importe (via `use`) le module `fmt` pour le rendre accessible.
use std::fmt;

// Nous définissons une structure dans laquelle le trait `fmt::Display`
// sera implémenté. Ce n'est qu'un simple tuple, nommée `Structure`, contenant un entier de type i32.
struct Structure(i32);

// Pour pouvoir utiliser le marqueur `{}`, le trait `fmt::Display` doit être implémenté
// manuellement pour le type.
impl fmt::Display for Structure {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // écrit le premier élément de la structure dans le flux en sortie
        // soumis: `f`. Renvoie une instance de `fmt::Result` qui témoigne du succès
        // ou de l'échec de l'opération. Notez que `write!` possède une syntaxe très
        // similaire à `println!`.
        write!(f, "{}", self.0)
    }
}
```

fmt::Display pourrait être plus lisible que fmt::Debug mais il présente un problème pour la bibliothèque standard. Comment les types ambiguës devraient être affichés ? Par exemple, si la bibliothèque standard devait implémenter un seul formatage pour toutes les « variantes » de Vec<T>, quel style devrait être choisi ? N'importe lequel ?

1. Vec<Path> : /:/etc:/home/username:/bin (séparé par des « : ») ;
2. Vec<i32> : 1,2,3 (séparé par des « , »).

Bien sûr que non, puisqu'il n'y a pas de mise en forme idéale pour tous les types et la bibliothèque standard n'en impose pas.

fmt::Display n'est pas implémenté pour la structure Vec<T> ni pour aucun autre conteneur générique. fmt::Debug doit alors être utilisé pour ces ressources.

Ce n'est en revanche pas un problème pour les conteneurs(e.g. structures) qui ne sont pas génériques, fmt::Display peut être implémenté et utilisé.

```
use std::fmt; // On importe le module `fmt`

// Une structure qui contient deux nombres. `Debug` va être hérité pour que les résultats
// puissent être comparés avec `Display`.
#[derive(Debug)]
struct MinMax(i64, i64);

// Implémentation du trait `Display` pour la structure `MinMax`.
impl fmt::Display for MinMax {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // On utilise `self.nombre` pour faire référence à la donnée se trouvant
        // à cette position.
        write!(f, "({}, {})", self.0, self.1)
    }
}

// Définissons une structure où les champs sont nommés pour comparer.
#[derive(Debug)]
struct Point2D {
    x: f64,
    y: f64,
}
```

```

// On implémente également le trait fmt::Display pour la structure Point2D
impl fmt::Display for Point2D {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // On désigne les champs de notre choix. (en l'occurrence `x` et `y`)
        write!(f, "x: {}, y: {}", self.x, self.y)
    }
}

fn main() {
    let minmax = MinMax(0, 14);

    println!("Comparaison des structures:");
    println!("Display: {}", minmax);
    println!("Debug: {:?}", minmax);

    let big_range =   MinMax(-300, 300);
    let small_range = MinMax(-3, 3);

    println!("Le grand intervalle est {big} et le petit est {small}",
             small = small_range,
             big = big_range);

    let point = Point2D { x: 3.3, y: 7.2 };

    println!("Comparaison des points:");
    println!("Display: {}", point);
    println!("Debug: {:?}", point);

    // Erreur. Les traits `Debug` and `Display` étaient implémentés mais
    // le marqueur `{:b}` requiert l'implémentation du trait `fmt::Binary`.
    // Cela ne fonctionnera pas.
    // println!("A quoi ressemble Point2D formaté en binaire: {:b} ?", point);
}
```

Donc fmt::Display a été implémenté mais ce n'est pas le cas de fmt::Binary, il ne peut alors pas être utilisé.

std::fmt possède de nombreux **traits** et chacun doit posséder sa propre implémentation. Pour plus d'informations, nous vous invitons à consulter **la documentation du module**.

Activité

Après avoir constaté le résultat de l'exemple ci-dessus, aidez-vous de la structure Point2D pour ajouter à l'exemple une nouvelle structure nommée Complex. Voici le résultat attendu lorsqu'une instance de la structure Complex sera affichée :

```
Display: 3.3 + 7.2i
Debug: Complex { real: 3.3, imag: 7.2 }
```

Voir aussi

L'attribut derive, **std::fmt**, **les macros**, **les structures**, **les traits**, **le mot-clé use**.

1-2-2-1 - Exemple d'utilisation : La structure List

Implémenter le trait fmt::Display pour une structure où les éléments doivent être gérés séquentiellement est assez délicat. Le problème réside dans le fait que chaque appel de la macro write! génère une instance de fmt::Result. Une bonne gestion de ces appels nécessite de tester chaque résultat. Rust vous permet de gérer les erreurs de deux manières :

1 En utilisant la macro try! ;
2 En utilisant l'opérateur ? (qui est l'équivalent de try! mais intégré directement au langage).

La macro try! vient envelopper la fonction (ou la macro) cible comme ceci :

```
// On 'test' write! Pour voir si une erreur survient. S'il y a une erreur,
// elle sera renvoyée. Sinon, l'exécution continue.
try!(write!(f, "{}", value));
```

L'opérateur ?, bien qu'équivalent à la macro try!, vient se positionner devant l'appel de la fonction (ou macro).

```
write!(f, "{}", value)?;
```

Avec l'opérateur ?, l'implémentation du trait fmt::Display pour un Vec est simple et lisible.

```
use std::fmt; // On importe le module `fmt`.


// On déclare une structure nommée 'List' qui contient un 'Vec'.
struct List(Vec<i32>);

impl fmt::Display for List {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        // On extrait le premier champ de la structure.
        // Nous créons une référence de 'vec'.
        let vec = &self.0;

        write!(f, "[")?;

        // On parcourt 'vec' en stockant chacun de ses éléments et
        // le nombre d'itérations.
        for (count, v) in vec.iter().enumerate() {
            // Pour tout élément, excepté le premier, on ajoute une virgule.
            // On utilise l'opérateur ?, ou la macro try!, pour renvoyer les erreurs.
            if count != 0 { write!(f, ", ")?; }
            write!(f, "{}", v)?;
        }

        // On ferme le crochet ouvert précédemment et on renvoie une instance
        // de la structure `fmt::Result`.
        write!(f, "]")
    }
}

fn main() {
    let v = List(vec![1, 2, 3]);
    println!("{}", v);
}
```

Activité

Essayez de modifier le programme pour que l'index de chaque élément du vector soit également affiché durant l'exécution. Le résultat devrait ressembler à ceci :

```
[0: 1, 1: 2, 2: 3]
```

Voir aussi

La boucle for, **le pattern ref**, **Result**, **les structures**, **try!**, **vec!**.

1-2-3 - Formatage

Nous avons vu que le formatage désiré était spécifié par des « chaînes de formatage » :

- format!("{}", foo) -> "3735928559" ;

- format!("0x{:X}", foo) -> "0xDEADBEEF" ;
- format!("0o{:o}", foo) -> "0o33653337357".

La même variable (foo) peut être formatée de différentes manières suivant le type d'argument utilisé dans le marqueur (e.g. X, o, rien).

Cette fonctionnalité est implémentée à l'aide de traits, et il y en a un pour chaque type d'argument. Le plus commun est, bien entendu, Display. Il est chargé de gérer les cas où le type d'argument n'est pas spécifié (i.e. {}).

```
use std::fmt::{self, Formatter, Display};

struct City {
    name: &'static str,
    // Latitude
    lat: f32,
    // Longitude
    lon: f32,
}

impl Display for City {
    // `f` est un tampon, cette méthode écrit la chaîne de caractères
    // formattée à l'intérieur de ce dernier.
    fn fmt(&self, f: &mut Formatter) -> fmt::Result {
        let lat_c = if self.lat >= 0.0 { 'N' } else { 'S' };
        let lon_c = if self.lon >= 0.0 { 'E' } else { 'W' };

        // `write!` est équivalente à `format!`, à l'exception qu'elle écrira
        // la chaîne de caractères formatée dans un tampon (le premier argument).
        write!(f, "{}: {:.3}°{} {:.3}°{}",
               self.name, self.lat.abs(), lat_c, self.lon.abs(), lon_c)
    }
}

#[derive(Debug)]
struct Color {
    red: u8,
    green: u8,
    blue: u8,
}

fn main() {
    for city in [
        City { name: "Dublin", lat: 53.347778, lon: -6.259722 },
        City { name: "Oslo", lat: 59.95, lon: 10.75 },
        City { name: "Vancouver", lat: 49.25, lon: -123.1 },
    ].iter() {
        println!("{}", *city);
    }
    for color in [
        Color { red: 128, green: 255, blue: 90 },
        Color { red: 0, green: 3, blue: 254 },
        Color { red: 0, green: 0, blue: 0 },
    ].iter() {
        // Utilisez le marqueur `{}` une fois que vous aurez implémenté
        // le trait fmt::Display.
        println!("{:?}", *color)
    }
}
```

N'hésitez pas à consulter **la liste complète des traits** dédiés au formatage ainsi que leurs types d'argument dans la documentation du module **std::fmt**.

Activité

Implémentez le trait fmt::Display pour la structure Color dans l'exemple ci-dessus de manière à obtenir un résultat identique à celui-ci :

```
RGB (128, 255, 90) 0x80FF5A
RGB (0, 3, 254) 0x0003FE
RGB (0, 0, 0) 0x000000
```

Indices :

- Vous pourriez **avoir besoin d'itérer plusieurs fois** sur vos couleurs ;
- Vous pouvez **créer une « compensation »** (remplissant votre chaîne de zéros) d'une largeur n avec :0n.

Voir aussi

std::fmt

2 - Les primitifs

Le langage Rust offre une grande variété de primitifs. Liste non-exhaustive :

- Les entiers signés : i8, i16, i32, i64 et isize (dépend de l'architecture de la machine) ;
- Les entiers non-signés : u8, u16, u32, u64, usize (dépend de l'architecture de la machine) ;
- Les réels : f32, f64 ;
- Les caractères (Unicode) : 'a', 'α', '∞'. Codés sur 4 octets ;
- Les booléens : true ou false ;
- L'absence de type (), qui n'engendre qu'une seule valeur : () ;
- Les tableaux : [1, 2, 3] ;
- Les tuples : (1, true).

Le type des variables peut toujours être spécifié. Les nombres peuvent également être typés grâce à un suffixe, ou par défaut (laissant le compilateur les typer). Les entiers, par défaut, sont typés i32 tandis les réels sont typés f64.

```
fn main() {
    // Le type des variables peut être spécifié, annoté.
    let logical: bool = true;

    let a_float: f64 = 1.0;  // typage classique
    let an_integer   = 5i32; // Typage par suffixe

    // Le type par défaut peut également être conservé.
    // typage implicite
     let default_float   = 3.0; // `f64`
     let default_integer = 7;   // `i32`

     let mut mutable = 12; // Entier signé codé sur 4 octets (i32).

     // Erreur! Le type d'une variable ne peut être modifié en cours de route.
     // mutable = true;
}
```

Voir aussi

La bibliothèque standard.

2-1 - Les littéraux et les opérateurs

Les entiers (1), les réels (1,2), les caractères ('a'), les chaînes de caractères ("abc"), les booléens (true) et l'absence de type () (un tuple vide) peuvent être représentés en utilisant les littéraux.

Les entiers peuvent également être exprimés sous différentes bases : hexadécimal, octal ou binaire en utilisant, respectivement, les préfixes : 0x, 0o ou 0b.

Des underscores peuvent être insérés à l'intérieur des littéraux numériques pour soigner la lisibilité (e.g. 1_000 est équivalent à 1000 et 0.000_001 est équivalent à 0.000001).

Nous devons renseigner le compilateur quant au type de littéral que nous utilisons. Pour le moment, nous allons utiliser le suffixe u32 pour indiquer que le littéral est un entier non-signé codé sur 32 bits et le suffixe i32 pour indiquer que c'est un entier signé codé sur 32 bits.

Les opérateurs et leur priorité **dans le langage Rust** peuvent être retrouvés dans **les langages « C-like »**.

```
fn main() {
    // Addition d'entiers
    println!("1 + 2 = {}", 1u32 + 2);

    // Soustraction d'entiers
    println!("1 - 2 = {}", 1i32 - 2);
  // TODO ^ Essayez de changer `1i32` par `1u32` pour prendre conscience de l'importance du type

    // Logique booléenne
     println!("true AND false is {}", true && false);
     println!("true OR false is {}", true || false);
     println!("NOT true is {}", !true);

     // Opérations bit-à-bit
     println!("0011 AND 0101 is {:04b}", 0b0011u32 & 0b0101);
     println!("0011 OR 0101 is {:04b}", 0b0011u32 | 0b0101);
     println!("0011 XOR 0101 is {:04b}", 0b0011u32 ^ 0b0101);
     println!("1 << 5 is {}", 1u32 << 5);
     println!("0x80 >> 2 is 0x{:x}", 0x80u32 >> 2);

     // Utilisez des underscores pour améliorer la lisibilité !
     println!("One million is written as {}", 1_000_000u32);
}
```

2-2 - Les tuples

Un tuple est une collection de valeurs de différents types (ou pas). Les tuples peuvent être construits en utilisant les parenthèses () et chaque tuple est lui-même un type possédant sa propre signature (T1, T2, ...), où T1, T2 sont les types de ses membres. Les fonctions peuvent se servir des tuples pour renvoyer plusieurs valeurs, puisque ces derniers peuvent être extensibles à volonté.

```
// Les tuples peuvent être utilisés comme arguments passés à une fonction
// et comme valeurs de renvoi.
fn reverse(pair: (i32, bool)) -> (bool, i32) {
    // `let` peut être utilisé pour assigner, lier les membres d'un tuple à des
    // variables.
    let (integer, boolean) = pair;

    (boolean, integer)
}

// La structure suivante est dédiée à l'activité.
#[derive(Debug)]
struct Matrix(f32, f32, f32, f32);

fn main() {
    // Un tuple composé de différents types
    let long_tuple = (1u8, 2u16, 3u32, 4u64,
                      -1i8, -2i16, -3i32, -4i64,
                      0.1f32, 0.2f64,
                      'a', true);

    // Les valeurs peuvent être extraites depuis le tuple en utilisant son
    // indexation.
    println!("long tuple first value: {}", long_tuple.0);
    println!("long tuple second value: {}", long_tuple.1);
```

```

    // Les tuples peuvent être eux-même des membres d'un tuple.
    let tuple_of_tuples = ((1u8, 2u16, 2u32), (4u64, -1i8), -2i16);

    // Les tuples peuvent être affichés avec Debug.
    println!("tuple of tuples: {:?}", tuple_of_tuples);

    let pair = (1, true);
    println!("pair is {:?}", pair);

    println!("the reversed pair is {:?}", reverse(pair));

    // Pour créer un élément de tuple, une virgule est requise pour différencier
    // un élément de tuple d'un simple litéral entouré de parenthèses.
    println!("one element tuple: {:?}", (5u32,));
    println!("just an integer: {:?}", (5u32));

    // Les tuples peuvent être "déstructurés" (i.e. décomposés pour créer de
    // nouvelles assignations).
    let tuple = (1, "hello", 4.5, true);

    let (a, b, c, d) = tuple;
    println!("{:?}, {:?}, {:?}, {:?}", a, b, c, d);

    let matrix = Matrix(1.1, 1.2, 2.1, 2.2);
    println!("{:?}", matrix)

}
```

Activité

1 Récapitulatif : Implémentez les services du trait fmt::Display pour la structure Matrix dans l'exemple ci-dessus. Donc si vous passez de l'affichage de débogage {:?} à l'affichage plus « user friendly » {}, vous devriez voir le résultat suivant :

```
( 1.1 1.2 )
( 2.1 2.2 )
```

Vous pouvez vous référer à l'exemple précédemment donné pour **l'implémentation du trait Display**.

1 Ajoutez une fonction transpose(), en vous appuyant sur l'implémentation de la fonction reverse(), qui accepte une matrice en paramètre et renvoie une matrice dans laquelle deux éléments ont été inversés. Exemple :

```
println!("Matrix:\n{}", matrix);
println!("Transpose:\n{}", transpose(matrix));
```

Affiche :

```
Matrix:
( 1.1 1.2 )
( 2.1 2.2 )
Transpose:
( 1.1 2.1 )
( 1.2 2.2 )
```

2-3 - Les tableaux et les "slices"

Un tableau est une collection d'objets appartenant au même type T, contenu dans un bloc mémoire défragmenté. Vous pouvez créer un tableau en utilisant les crochets [] et leur taille, connue à la compilation, fait partie intégrante de la signature du type [T; taille].



***Note** : Le terme « slice », en français, pourrait être traduit par « morceau », « tranche », g. Pour la suite du chapitre, nous utiliserons le terme « slice ».*

Les slices sont similaires aux tableaux, à l'exception de leur taille qui n'est pas connue à la compilation. Une slice est un objet composé de deux « mots », le premier étant un pointeur vers la ressource initiale et le second étant la taille de la slice. La taille en mémoire de la slice est déterminée par l'architecture du processeur (e.g. **64 bits** pour une architecture **x86-64**). Les slices peuvent être utilisées pour isoler une partie d'un tableau et héritent de la signature de ce dernier &[T].

```
use std::mem;


// Cette fonction emprunte une slice.
fn analyze_slice(slice: &[i32]) {
    println!("first element of the slice: {}", slice[0]);
    println!("the slice has {} elements", slice.len());
}

fn main() {
    // Tableau dont la taille est connue à la compilation (le type peut être omis).
    let xs: [i32; 5] = [1, 2, 3, 4, 5];

    // Tous les éléments peuvent être initialisés à la même valeur.
    let ys: [i32; 500] = [0; 500];

    // L'indexation débute à 0.
    println!("first element of the array: {}", xs[0]);
    println!("second element of the array: {}", xs[1]);

    // `len` renvoie la taille du tableau.
    println!("array size: {}", xs.len());

    // Les tableaux sont alloués dans la pile.
    println!("array occupies {} bytes", mem::size_of_val(&xs));

    // Les tableaux peuvent être automatiquement empruntés en tant que
    // slice.
    println!("borrow the whole array as a slice");
    analyze_slice(&xs);

    // Les slices peuvent pointer sur une partie bien précise d'un tableau.
    println!("borrow a section of the array as a slice");
    analyze_slice(&ys[1 .. 4]);

    // Erreur! Le dépassement de tampon fait planter le programme.
    // println!("{}", xs[5]);
}
```

3 - Les types personnalisés

En Rust, les types de données personnalisés sont principalement créés à partir de ces deux mots-clés :

1. struct : Définit une structure ;
2. enum : Définit une énumération.

Les constantes peuvent également être créées via les mots-clés const et static.

3-1 - Les structures

Il y a trois types de structures pouvant être créé en utilisant le mot-clé struct :

1. Les « tuple structs », aussi appelées simplement tuples ;
2. Les **structures classiques** issues du langage C ;
3. Les structures unitaires. Ne possèdant aucun champ, elles sont utiles pour la généricité.

```
// Une structure unitaire.
struct Nil;

// Un tuple.
struct Pair(i32, f32);

// Une structure avec deux champs.
struct Point {
    x: f32,
    y: f32,
}

// Les structures peuvent faire partie des champs d'une autre structure.
#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}

fn main() {
    // On instancie la structure `Point`.
    let point: Point = Point { x: 0.3, y: 0.4 };

    // On accède aux champs du point.
    println!("point coordinates: ({}, {})", point.x, point.y);

    // On décompose les champs de la structure pour les assigner
    // à de nouvelles variables (i.e. my_x et my_y)
    let Point { x: my_x, y: my_y } = point;

    let _rectangle = Rectangle {
        // L'instanciation de la structure est également une expression.
        p1: Point { x: my_y, y: my_x },
        p2: point,
    };

    // On instancie la structure unitaire, vide.
    let _nil = Nil;

    // On instancie un tuple.
    let pair = Pair(1, 0.1);

    // Accède aux champs du tuple.
    println!("pair contains {:?} and {:?}", pair.0, pair.1);

    // On décompose un tuple.
    let Pair(integer, decimal) = pair;

    println!("pair contains {:?} and {:?}", integer, decimal);
}
```

Activité

1. Ajoutez une fonction `rect_area` qui calcule l'air d'un rectangle (essayez d'utiliser la déstructuration) ;
2. Ajoutez une fonction `square` qui prend en paramètre une instance de la structure `Point` et un réel de type `f32` puis renvoie une instance de la structure `Rectangle` contenant le point du coin inférieur gauche du rectangle ainsi qu'une largeur et une hauteur correspondant au réel passé en paramètre à la fonction `square`.

Voir aussi

Les attributs et **la déstructuration**.

3-2 - Les énumérations

Le mot-clé enum permet la création d'un type qui peut disposer d'une ou plusieurs variantes de lui-même. Toutes **les variantes des structures** sont valides dans une énumération.

```
// Masque les avertissements du compilateur lorsqu'il y a
// du code mort présent dans votre code.
#![allow(dead_code)]

// On créé une énumération pour définir des "classes" de personnes.
// Notez que chaque variante de l'énumération est indépendante de l'autre.
// Aucune n'est égale à l'autre: `Engineer != Scientist` et
// `Height(i32) != Weight(i32)`.
enum Person {
    // Une variante peut être une structure unitaire,
    Engineer,
    Scientist,
    // un tuple
    Height(i32),
    Weight(i32),
    // ou simplement une structure classique.
    Info { name: String, height: i32 }
}

// Prend une variante de l'énumération `Person` en argument et
// ne renvoie rien.
fn inspect(p: Person) {
    // En utilisant une énumération, vous devez analyser tous les cas
    // possibles (obligatoire).
    // Le pattern matching permet de les couvrir efficacement.
    match p {
        Person::Engineer  => println!("Is an engineer!"),
        Person::Scientist => println!("Is a scientist!"),
        // On récupère l'attribut de l'instance `Height`.
        Person::Height(i) => println!("Has a height of {}.", i),
        Person::Weight(i) => println!("Has a weight of {}.", i),
        // Destructure `Info` into `name` and `height`.
        // On récupère les attributs
        Person::Info { name, height } => {
            println!("{} is {} tall!", name, height);
        },
    }
}

fn main() {
    let person   = Person::Height(18);
    let amira    = Person::Weight(10);
    // La fonction `to_owned()` créé une instance de la structure `String`
    // possédée par l'assignation `name` à partir d'une slice (i.e. &str).
    let dave     = Person::Info { name: "Dave".to_owned(), height: 72 };
    let rebecca  = Person::Scientist;
    let rohan    = Person::Engineer;

    inspect(person);
    inspect(amira);
    inspect(dave);
    inspect(rebecca);
    inspect(rohan);
}
```

Voir aussi

Les attributs, **le mot-clé match**, **le mot-clé fn**, **les chaînes de caractères**.

3-2-1 - Le mot-clé use

Grâce au mot-clé use, il n'est pas toujours obligatoire de spécifier le contexte d'une ressource à chaque utilisation.

```
// Masque les avertissements du compilateur concernant le code mort.
#![allow(dead_code)]

enum Status {
    Rich,
    Poor,
}

enum Work {
    Civilian,
    Soldier,
}

fn main() {
    // Nous précisons que ces variantes de l'énumération sont utilisées, donc
    // il n'est plus nécessaire de préciser leur conteneur.
    use Status::{Poor, Rich};
    // On utilise automatiquement toutes les variantes de l'enum `Work`.
    use Work::*;

    // Equivalent à `Status::Poor`.
    let status = Poor;
    // Equivalent à `Work::Civilian`.
    let work = Civilian;

    match status {
        // Notez la disparition du conteneur lors de la recherche de pattern.
        Rich => println!("The rich have lots of money!"),
        Poor => println!("The poor have no money..."),
    }

    match work {
        // Une fois encore, le conteneur a disparu.
        Civilian => println!("Civilians work!"),
        Soldier  => println!("Soldiers fight!"),
    }
}
```

Voir aussi

Le mot-clé match et **la déclaration use**.

3-2-2 - Énumérations "C-like"

Les énumérations du langage Rust peuvent également adopter la même syntaxe que celles du langage C (possédant un identifiant explicite).

```
// Un attribut qui masque les avertissements du compilateur
// concernant le code mort.
#![allow(dead_code)]

// Énumération avec un identifiant implicite (partant de 0).
enum Number {
    Zero, // 0
    One, // 1
    Two, // 2
}

// Énumération avec un identifiant explicite.
enum Color {
    Red = 0xff0000,
    Green = 0x00ff00,
```

```
    Blue = 0x0000ff,
}

fn main() {
    // Les variantes d'une énumération peuvent être converties en entiers.
    println!("zero is {}", Number::Zero as i32);
    println!("one is {}", Number::One as i32);

    println!("roses are #{:06x}", Color::Red as i32);
    println!("violets are #{:06x}", Color::Blue as i32);
}
```

Voir aussi

Le casting.

3-2-3 - Exemple d'utilisation : « linked-list »

Voici un exemple dans lequel une énumération peut être utilisée pour créer une liste de nœuds :

```
use List::*;

enum List {
    // Cons: Un tuple contenant un élément(i.e. u32) et un pointeur vers le noeud suivant (i.e. Box<List>).
    Cons(u32, Box<List>),
    // Nil: Un noeud témoignant de la fin de la liste.
    Nil,
}

// Il est possible de lier, d'implémenter des méthodes
// pour une énumération.
impl List {
    // Créé une liste vide.
    fn new() -> List {
        // `Nil` est une variante de `List`.
        Nil
    }

    // Consomme, s'approprie la liste et renvoie une copie de cette même liste
    // avec un nouvel élément ajouté à la suite.
    fn prepend(self, elem: u32) -> List {
        // `Cons` est également une variante de `List`.
        Cons(elem, Box::new(self))
    }

    // Renvoie la longueur de la liste.
    fn len(&self) -> u32 {
        // `self` doit être analysé car le comportement de cette méthode
        // dépend du type de variante auquel appartient `self`.
        // `self` est de type `&List` et `*self` est de type `List`, rendant
        // possible l'analyse directe de la ressource plutôt que par le biais d'un alias (i.e. une référence).
        // Pour faire simple: on déréférence `self` avant de l'analyser.
        // Note: Lorsque vous travaillez sur des références, préférez le déréférencement
        // avant analyse.
        match *self {
            // On ne peut pas prendre "l'ownership" de la queue (liste)
            // puisque l'on emprunte seulement `self` (nous ne le possédons pas);
            // Nous créerons simplement une référence de la queue.
            Cons(_, ref tail) => 1 + tail.len(),
            // De base, une liste vide possède 0 élément.
            Nil => 0
        }
    }

    // Renvoie une représentation de la liste sous une chaîne de caractères
    // (wrapper)
```

```
    fn stringify(&self) -> String {
        match *self {
            Cons(head, ref tail) => {
                // `format!` est équivalente à `println!` mais elle renvoie
                // une chaîne de caractères allouée dans le tas (wrapper)
                // plutôt que de l'afficher dans la console.
                format!("{}, {}", head, tail.stringify())
            },
            Nil => {
                format!("Nil")
            },
        }
    }
}

fn main() {
    // Créé une liste vide.
    let mut list = List::new();

    // On ajoute quelques éléments.
    list = list.prepend(1);
    list = list.prepend(2);
    list = list.prepend(3);

    // Affiche l'état définitif de la liste.
    println!("La linked list possède une longueur de: {}", list.len());
    println!("{}", list.stringify());
}
```

Voir aussi

Box, **les méthodes**.

3-3 - Les constantes

Rust possède deux types de constantes qui peuvent être déclarées dans n'importe quel contexte global.

Chacun dispose d'un mot-clé :

- const : Une valeur immuable (état par défaut de toute variable) ;
- static : Une variable pouvant être accédée en lecture et (accessoirement) en écriture possédant la « lifetime » 'static.

Exception pour les "chaînes de caractères" littérales qui peuvent être directement assignées à une variable statique sans modification de votre part, car leur type &'static str dispose déjà de la lifetime 'static. Tous les autres types de référence doivent être explicitement annotés pour étendre leur durée de vie.

```
// Les variables globales sont déclarées en dehors de tous contextes.
static LANGUAGE: &'static str = "Rust";
const  THRESHOLD: i32 = 10;

fn is_big(n: i32) -> bool {
    // Accès à la constante dans une fonction.
    n > THRESHOLD
}

fn main() {
    let n = 16;

    // Accès à la constante dans le fil d'exécution principal.
    println!("This is {}", LANGUAGE);
    println!("The threshold is {}", THRESHOLD);
    println!("{} is {}", n, if is_big(n) { "big" } else { "small" });

    // Erreur! Vous ne pouvez pas modifier une constante.
```

```
    // THRESHOLD = 5;
    // FIXME ^ Commentez cette ligne pour voir disparaître
    // le message d'erreur.
}
```

Voir aussi

La RFC des mot-clés **const et static**, **la lifetime 'static**.

4 - Les assignations

Rust assure l'immuabilité du type d'une variable grâce au typage statique. Lorsqu'une variable est déclarée elle peut être typée. Cependant, dans la plupart des cas, le compilateur sera capable d'inférer le type de la variable en se basant sur le contexte, atténuant sérieusement la lourdeur du typage.

Les valeurs (tels que les littéraux) peuvent être assignées à des variables en utilisant le mot-clé `let`.

```
fn main() {
    let an_integer = 1u32;
    let a_boolean = true;
    let unit = ();

    // Copie `an_integer` dans `copied_integer`.
    let copied_integer = an_integer;

    println!("An integer: {:?}", copied_integer); // un entier
    println!("A boolean: {:?}", a_boolean); // un booléen
    println!("Meet the unit value: {:?}", unit); // rien

    // Le compilateur vous alertera lorsqu'il détecte une variable inutilisée;
    // Vous pouvez faire taire ces avertissements en préfixant l'identificateur
    // de la variable avec un underscore (i.e. _).
    let _unused_variable = 3u32;

    let noisy_unused_variable = 2u32;
    // FIXME ^ Préfixez cet identificateur avec un underscore pour supprimer
    // l'avertissement.
}
```

4-1 - Mutabilité

L'assignation à une variable est immuable par défaut mais ceci peut être changé en utilisant le modificateur `mut`.

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Avant modification: {}", mutable_binding);

    // Ok
    mutable_binding += 1;

    println!("Après modification: {}", mutable_binding);

    // Erreur!
    //_immutable_binding += 1;
    // FIXME ^ Décommentez cette ligne pour voir le message d'erreur
}
```

4-2 - Scope et shadowing

Les assignations possèdent un contexte dans lequel elles persisteront et qui sera représenté par un « bloc ». Un bloc est une suite d'instructions et de déclarations englobées par des accolades {}. Le **shadowing** est également permis.

```
fn main() {
    // Cette assignation vit dans la fonction `main`.
    let long_lived_binding = 1;

    // Ceci est un bloc, il possède un contexte plus petit que celui de la fonction
    // `main`.
    {
        // Cette assignation existe seulement dans ce bloc.
        let short_lived_binding = 2;

         println!("inner short: {}", short_lived_binding);

         // Cette assignation *masque* l'assignation du contexte supérieur (la fonction
         // `main`).
         let long_lived_binding = 5_f32;

         println!("inner long: {}", long_lived_binding);
     }
     // Fin du bloc.

     // Erreur! `short_lived_binding` n'existe pas dans ce contexte.
     // println!("outer short: {}", short_lived_binding);
     // FIXME ^ Décommentez cette ligne pour voir l'erreur.

     println!("outer long: {}", long_lived_binding);

     // Cette assignation *masque* également l'assignation précédente.
     let long_lived_binding = 'a';

     println!("outer long: {}", long_lived_binding);
 }
```

4-3 - Déclaration seule

Il est possible de déclarer une variable dans un premier temps, pour l'initialiser dans un second temps. Cependant, cette forme est rarement utilisée puisqu'elle peut conduire à l'utilisation de variables qui ne sont pas initialisées (et donc à faire des erreurs).

```
fn main() {
    // On déclare une variable.
    let a_binding;

    {
        let x = 2;

        // On initialise la variable.
        a_binding = x * x;
     }

     println!("a binding: {}", a_binding);

     let another_binding;

     // Erreur! Utilisation d'une variable non-initialisée.
     // println!("another binding: {}", another_binding);
     // FIXME ^ Décommentez cette ligne pour voir l'erreur.
     another_binding = 1;

     println!("another binding: {}", another_binding);
 }
```

Comme l'utilisation d'une variable, qui n'a pas été initialisée au préalable, peut mener à des comportements imprévisibles à l'exécution, le compilateur vous interdit de les utiliser.

5 - Le casting

Rust ne permet pas la conversion implicite des types primitifs (coercition). En revanche, une conversion explicite (casting) peut être entreprise à l'aide du mot-clé `as`.

Les règles régissant la conversion entre les types littéraux s'inspirent, principalement, des **conventions du langage C** à l'exception des cas où le C réserve des comportements imprévisibles.

```rust
// Supprime tous les avertissements relatifs aux dépassements
// de capacité (e.g. une variable de type u8 ne peut pas
// contenir plus qu'une variable de type u16).
#![allow(overflowing_literals)]

fn main() {
    let decimal = 65.4321_f32;

    // Erreur! La conversion implicite n'est pas supportée.
    // let integer: u8 = decimal;
    // FIXME ^ Décommentez/Commentez cette ligne pour voir
    // le message d'erreur apparaître/disparaître.

    // Conversion explicite.
    let integer = decimal as u8;
    let character = integer as char;

    println!("Casting: {} -> {} -> {}", decimal, integer, character);

    // Lorsque vous convertissez une valeur vers un type
    // non-signé T, std::T::MAX + 1 est incrémenté ou soustrait jusqu'à
    // ce que la valeur respecte la capacité du nouveau type.

    // 1000 ne dépasse pas la capacité d'un entier non-signé codé sur 16 bits.
    println!("1000 as a u16 is: {}", 1000 as u16);

    // 1000 - 256 - 256 - 256 = 232
    // En réalité, les 8 premiers bits les plus faibles (LSB) sont conservés et les
    // bits les plus forts (MSB) restants sont tronqués.
    println!("1000 as a u8 is : {}", 1000 as u8);
    // -1 + 256 = 255
    println!("  -1 as a u8 is : {}", (-1i8) as u8);

    // Pour les nombres positifs, cette soustraction est équivalente à une
    // division par 256.
    println!("1000 mod 256 is : {}", 1000 % 256);

    // Quand vous convertissez un type d'entiers signés, le résultat (bit à bit)
    // est équivalent à celui de la conversion vers un type d'entiers non-signés.
    // Si le bit de poids fort vaut 1, la valeur sera négative.

    // Sauf si il n'y a pas de dépassements, évidemment.
    println!(" 128 as a i16 is: {}", 128 as i16);
    // 128 as u8 -> 128, complément à deux de 128 codé sur 8 bits:
    println!(" 128 as a i8 is : {}", 128 as i8);

    // On répète l'exemple ci-dessus.
    // 1000 as u8 -> 232
    println!("1000 as a i8 is : {}", 1000 as i8);
    // et le complément à deux de 232 est -24.
    println!(" 232 as a i8 is : {}", 232 as i8);


}
```

5-1 - Les littéraux

Les littéraux numériques peuvent être typés en suffixant le littéral avec son type. Par exemple, pour préciser que le littéral 42 devrait posséder le type i32, nous écrirons 42i32.

Le type des littéraux numériques qui ne sont pas suffixés va dépendre du contexte dans lequel ils sont utilisés. S'il n'y a aucune contrainte (i.e. si rien ne force la valeur à être codée sur un nombre de bits bien précis), le compilateur utilisera le type i32 pour les entiers et f64 pour les nombres réels.

```
fn main() {
    // Ces littéraux sont suffixés, leurs types sont connus à l'initialisation.
    let x = 1u8;
    let y = 2u32;
    let z = 3f32;

    // Ces littéraux ne sont pas suffixés, leurs types dépendent du contexte.
    let i = 1;
    let f = 1.0;

     // La fonction `size_of_val` renvoie la taille d'une variable en octets.
     println!("La taille de `x` en octets: {}", std::mem::size_of_val(&x));
     println!("La taille de `y` en octets: {}", std::mem::size_of_val(&y));
     println!("La taille de `z` en octets: {}", std::mem::size_of_val(&z));
     println!("La taille de `i` en octets: {}", std::mem::size_of_val(&i));
     println!("La taille de `f` en octets: {}", std::mem::size_of_val(&f));
}
```

Certains concepts présentés dans l'exemple ci-dessus n'ont pas encore été abordés. Pour les plus impatients, voici une courte explication :

- fun(&foo) : Cette syntaxe représente le passage d'un paramètre par référence plutôt que par valeur (i.e. fun(foo)). Pour plus d'informations, voir **le chapitre du système d'emprunts**.
- std::mem::size_of_val est une fonction mais appelée avec son chemin absolu. Le code peut être divisé et organisé en plusieurs briques logiques nommées *modules*. Pour le cas de la fonction size_of_val, elle se trouve dans le module mem, lui-même se trouvant dans le paquet std. Pour plus d'informations voir **les modules** et/ou **les « crates »**.

5-2 - L'inférence des types

Le moteur dédié à l'inférence des types est très intelligent. Il fait bien plus que d'inférer le type d'une r-value à l'initialisation. Il se charge également d'analyser l'utilisation de la variable dans la suite du programme pour inférer son type définitif. Voici un exemple plus avancé dédié à l'inférence :

```
fn main() {
    // Dû à l'annotation(suffixe), le compilateur sait que `elem` possède le type
    // u8.
    let elem = 5u8;

    // Crée un vecteur vide (un tableau dont la taille n'est pas définie).
    let mut vec = Vec::new();
    // À ce niveau, le compilateur ne connaît pas encore le type exact de `vec`,
    // il sait simplement que c'est un vecteur de quelque chose (`Vec<_>`).

     // On ajoute `elem` dans le vecteur.
     vec.push(elem);
     // Tada! Maintenant le compilateur sait que `vec` est un vecteur
     // d'entiers non-signés typés `u8` (`Vec<u8>`).
     // TODO ^ Essayez de commenter la ligne où se trouve `vec.push(elem)`.

     println!("{:?}", vec);
}
```

Aucun typage explicite n'était nécessaire, le compilateur est heureux et le programmeur aussi !

5-3 - Les alias

Le mot-clé type peut être utilisé pour donner un nouveau nom à un type existant. Les types doivent respecter la convention de nommage *CamelCase* ou le compilateur vous renverra un avertissement. L'exception à cette règle sont les types primitifs : usize, f32, etc.

```
// `NanoSecond` est le nouveau nom de `u64`.
type NanoSecond = u64;
type Inch = u64;

// Utilisons un attribut pour faire taire les
// avertissements.
#[allow(non_camel_case_types)]
type u64_t = u64;
// TODO ^ Essayez de supprimer l'attribut.

fn main() {
    // `NanoSecond` = `Inch` = `u64_t` = `u64`.
    let nanoseconds: NanoSecond = 5 as u64_t;
    let inches: Inch = 2 as u64_t;

    // Notez que les alias de types ne fournissent aucune sécurité supplémentaire,
    // car ce ne sont pas de nouveaux types (i.e. vous pouvez très bien changer
    // Inch par NanoSecond, vous n'aurez aucune erreur).
    println!("{} nanoseconds + {} inches = {} unit?",
             nanoseconds,
             inches,
             nanoseconds + inches);
}
```

Voir aussi

Les attributs.

6 - Les expressions

Un programme écrit en Rust est (principalement) composé d'une série de déclarations :

```
fn main() {
    // déclaration
    // déclaration
    // déclaration
}
```

Il y a plusieurs sortes de déclarations en Rust. Les deux plus communes sont les assignations et les expressions suivies par un point-virgule « ; » :

```
fn main() {
    // assignation
    let x = 5;

    // expression;
    x;
    x + 1;
    15;
}
```

Les blocs sont également des expressions, donc ils peuvent être utilisés comme r-value dans les assignations. La dernière expression dans le bloc sera assignée à la l-value. Notez toutefois que si la dernière expression du bloc se termine par un point-virgule « ; », la valeur de renvoi sera ().

```
fn main() {
    let x = 5u32;

    let y = {
        let x_squared = x * x;
        let x_cube = x_squared * x;

        // Cette expression sera assignée à `y`.
        x_cube + x_squared + x
     };

     let z = {
         // Le point-virgule supprime cette expression et `()` est assigné
         // à `z`.
         2 * x;
     };

     println!("x is {:?}", x);
     println!("y is {:?}", y);
     println!("z is {:?}", z);
}
```

7 - Contrôle du flux

La caractéristique commune à tous langages est la capacité à contrôler le flux : if/else, for, etc., et Rust ne fait pas exception. Allons voir ça !

7-1 - If/else

Les branchements conditionnels tels que if ou else sont similaires à d'autres langages. Contrairement à beaucoup d'entre-eux, la condition booléenne peut toutefois ne pas être enveloppée de parenthèses et chaque condition est suivie d'un bloc. Les conditions if/else sont des expressions et toutes les branches doivent renvoyer le même type.

```
fn main() {
    let n = 5;

    if n < 0 {
        print!("{} est négatif.", n);
    } else if n > 0 {
        print!("{} est positif.", n);
    } else {
        print!("{} est nul.", n);
     }

     let big_n =
         if n < 10 && n > -10 {
             println!(" et est un petit nombre, multiplions-le par dix");

             // Cette expression renvoie un entier de type `i32`.
             10 * n
         } else {
             println!(" est un grand nombre, divisons-le par deux");

             // Cette expression doit également renvoyer un entier de type `i32`.
             n / 2
             // TODO ^ Essayez de supprimer cette expression en ajoutant un point-virgule.
         };
     //   ^ Ne pas oubliez de mettre un point-virgule ici! Toutes les
     // assignations (`let`) doivent se terminer par un point-virgule.

     println!("{} -> {}", n, big_n);
}
```

7-2 - Le mot-clé loop

Rust fournit le mot-clé loop pour créer une boucle infinie.

Le mot-clé break peut être utilisé pour sortir de la boucle n'importe où tandis que le mot-clé continue peut être utilisé pour ignorer le reste de l'itération en cours et en débuter une nouvelle.

```
fn main() {
    let mut count = 0u32;

    println!("Comptons jusqu'à l'infini!");

    // Boucle infinie.
    loop {
        count += 1;

        if count == 3 {
            println!("trois");

            // Ignore le reste de l'itération.
            continue;
        }

        println!("{}", count);

        if count == 5 {
            println!("Ok, ça suffit!");

            // Sort de la boucle.
            break;
        }
    }
}
```

7-2-1 - L'imbrication et les labels

Il est possible de sortir (i.e. break) ou de relancer (i.e. continue) l'itération d'une boucle à partir d'une autre boucle interne à cette dernière. Pour ce faire, les boucles concernées doivent être annotées avec un 'label et il devra être passé aux instructions break et/ou continue.

```
#![allow(unreachable_code)] // permet de faire taire les avertissements
// relatifs au code mort.

fn main() {
    'externe: loop {
        println!("Entré dans la boucle annotée 'externe.");

        'interne: loop {
            println!("Entré dans la boucle annotée 'interne.");

            // Cette instruction nous ferait simplement
            // sortir de la boucle 'interne.
            // break;

            // On sort de la boucle 'externe
            // à partir de la boucle 'interne.
            break 'externe;
        }

        println!("Cette ligne ne sera jamais exécutée.");
    }

    println!("Sorti de la boucle annotée 'externe.");
}
```

7-3 - La boucle while

Le mot-clé while peut être utilisé pour itérer jusqu'à ce qu'une condition soit remplie.

Écrivons les règles de l'infâme **FizzBuzz** en utilisant une boucle while :

```
fn main() {
    // Un compteur.
    let mut n = 1;

    // Itère sur `n` tant que sa valeur est strictement inférieure
    // à 101.
    while n < 101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
         } else if n % 3 == 0 {
            println!("fizz");
         } else if n % 5 == 0 {
            println!("buzz");
         } else {
            println!("{}", n);
         }

         // Incrémente le compteur.
         n += 1;
     }
}
```

7-4 - La boucle for et les invervalles

L'ensemble for in peut être utilisé pour itérer à l'aide d'une instance Iterator. L'une des manières les plus simples pour créer un itérateur est d'utiliser la notation d'intervalle (« range notation ») a..b. Soit un intervalle [a;b[(comprend toutes les valeurs entre a (inclut) et b (exclut)).

Écrivons les règles de *FizzBuzz* en utilisant la boucle for au lieu de while.

```
fn main() {
    // `n` prendra pour valeur: 1, 2, ..., 100  au fil des itérations.
    for n in 1..101 {
        if n % 15 == 0 {
            println!("fizzbuzz");
        } else if n % 3 == 0 {
            println!("fizz");
        } else if n % 5 == 0 {
            println!("buzz");
         } else {
            println!("{}", n);
         }
     }
}
```

Voir aussi

Les itérateurs.

7-5 - Le pattern matching

Rust fournit le *pattern matching* via le mot-clé match, lequel peut être utilisé comme le mot-clé switch avec le langage C.

```
fn main() {
```

```
    let number = 13;
    // TODO ^ Assignez différentes valeurs à `number`.

    println!("Nature de number {}", number);
    match number {
        // Teste une seule valeur.
        1 => println!("Un!"),
        // Teste plusieurs valeurs.
         2 | 3 | 5 | 7 | 11 => println!("C'est un nombre premier."),
         // Teste l'intervalle [13;19].
         13...19 => println!("A teen"),
         // Couvre tous les autres cas.
         _ => println!("Ain't special"),
     }

     let boolean = true;
     // L'analyse est également une expression.
     let binary = match boolean {
         // Les branches du match doivent couvrir tous les cas possibles.
         false => 0,
         true => 1,
         // TODO ^ Essayez de commenter l'une de ses branches.
     };

     println!("{} -> {}", boolean, binary);
}
```

7-5-1 - La déstructuration

Un bloc match peut décomposer des objets de différentes manières.

7-5-1-1 - Les tuples

Les tuples peuvent être déstructurés (entendez « décortiqués », « décomposés ») dans un bloc match comme suit :

```
fn main() {
    let pair = (0, -2);
    // TODO ^ Essayez de modifier les valeurs contenues par
    // le tuple.

    println!("Dites m'en plus à propos de {:?}", pair);
    // match peut être utilisé pour déstructurer un tuple.
    match pair {
        // Déstructure `y`.
         (0, y) => println!("Le premier élément est égal à `0`
         et `y` égal à `{:?}`", y),
         (x, 0) => println!("`x` est égal à `{:?}` et le dernier est égal à `0`", x),
         _      => println!("Peu importe ce qu'ils sont."),
         // L'underscore `_` siginifie que vous ne souhaitez pas
         // assigner de valeurs à une variable, que vous souhaitez couvrir tous les
         // autres cas.
     }
}
```

Voir aussi

Les tuples.

7-5-1-2 - Les énumérations

Une énumération est déstructurée de la même manière :

```
// On fait taire les avertissements (puisqu'on utilise
```

```
// qu'une seule variante).
#[allow(dead_code)]
enum Color {
    // Identification implicite.
    Red,
    Blue,
    Green,
    // Ces variantes assignent plusieurs tuples sous différents noms: les modèles
     // de couleur.
     RGB(u32, u32, u32),
     HSV(u32, u32, u32),
     HSL(u32, u32, u32),
     CMY(u32, u32, u32),
     CMYK(u32, u32, u32, u32),
}

fn main() {
     let color = Color::RGB(122, 17, 40);
     // TODO ^ Essayez de modifier les valeurs du tuple.
     println!("De quelle couleur s'agit-il?");
     // Une énumération peut être déstructurée en utilisant le pattern matching.
     match color {
         Color::Red   => println!("La couleur rouge!"),
         Color::Blue  => println!("La couleur bleu!"),
         Color::Green => println!("La couleur vert!"),
         Color::RGB(r, g, b) =>
             println!("Rouge: {}, Vert: {}, et Bleu: {}!", r, g, b),
         Color::HSV(h, s, v) =>
             println!("Teinte: {}, Saturation: {}, Valeur: {}!", h, s, v),
         Color::HSL(h, s, l) =>
             println!("Teinte: {}, Saturation: {}, Lumière: {}!", h, s, l),
         Color::CMY(c, m, y) =>
             println!("Cyan: {}, Magenta: {}, Jaune: {}!", c, m, y),
         Color::CMYK(c, m, y, k) =>
             println!("Cyan: {}, Magenta: {}, Jaune: {}, Noir: {}!",
                 c, m, y, k),
         // Inutile d'ajouter une branche "par défaut" car tous les
         // cas ont été couverts.
     }
}
```

Voir aussi

L'attribut allow(...), les modèles de couleur **FR** ou **EN** et **les énumérations**.

7-5-1-3 - Les pointeurs et références

À propos des pointeurs, la distinction doit être faite entre la déstructuration et le déréférencement puisque ce sont deux concepts différents utilisés différemment par rapport au langage C.

- Le déréférencement utilise `*` ;
- La déstructuration utilise `&`, `ref`, et `ref mut`.

```
fn main() {
    // Assigne une référence de type `i32`. Le `&` signifie qu'une
    // référence est assignée.
    // L'équivalent non-raccourci de cette assignation pourrait ressembler à ceci:
    // ```rust
    // let _reference: i32 = 4;
    // let reference: &i32 = &_reference;
    // ```
    let reference: &i32 = &4;

     match reference {
         // Lorsque `reference` est comparé à `&val`, la comparaison ressemble à ceci:
         // `&i32`
         // `&val` <- `val` est plus ou moins une représentation de `reference`.
```

```
        // ^ Nous remarquons que si le `&` est omis, la valeur devrait être
        // assignée à `val`.
        &val => println!("On récupère une valeur via déstructuration: {:?}", val),
    }

    // Pour éviter d'utiliser la référence, vous pouvez déréférencer `reference`
    // avant analyse (vous permettant d'opérer sur la valeur, si elle est mutable).
    match *reference {
        val => println!("On récupère la valeur déréférencée: {:?}", val),
    }

    // Que se passe-t-il si vous ne créez pas une référence ? `reference`
    // était une référence parce que la r-value était une référence. Cette
    // variable n'en est pas une parce que la r-value n'en est pas une.
    let _not_a_reference = 3;

    // Rust fournit le mot-clé `ref` dans ce but. Il modifie l'assignation
    // de manière à créer une référence pour l'élément; cette référence est assignée.
    let ref _is_a_reference = 3;

    // Bien entendu, en assignant deux valeurs sans références, ces dernières
    // peuvent être récupérées à l'aide du mot-clé `ref` et `ref mut`.
    let value = 5;
    let mut mut_value = 6;

    // On utilise le mot-clé `ref` pour créer une référence.
    match value {
        ref r => println!("On récupère une référence de la valeur: {:?}", r),
    }

    // `ref mut` s'utilise de la même manière.
    match mut_value {
        ref mut m => {
            // On obtient une référence. Nous allons déréférencer `m` avant
            // de pouvoir opérer.
            *m += 10;
            println!("Nous incrémentons de 10. `mut_value`: {:?}", m);
        },
    }
}
```

7-5-1-4 - Les structures

Une structure peut également être déstructurée comme suit :

```
fn main() {
    struct Foo { x: (u32, u32), y: u32 }

    // Déstructure les membres de la structure.
    let foo = Foo { x: (1, 2), y: 3 };
    let Foo { x: (a, b), y } = foo;

   println!("a = {}, b = {},  y = {} ", a, b, y);

    // Vous pouvez déstructurer les structures et renommer
    // leurs variables. L'ordre n'est pas important.

    let Foo { y: i, x: j } = foo;
    println!("i = {:?}, j = {:?}", i, j);

    // et vous pouvez aussi ignorer certaines variables:
    let Foo { y, .. } = foo;
    println!("y = {}", y);

    // Ceci donne une erreur: le pattern ne mentionne pas le champ `x`.
    // let Foo { y } = foo;
}
```

Voir aussi

Les structures, **ref**.

7-5-2 - Les gardes

Lorsque vous usez du *pattern matching*, un « garde » peut être ajouté dans chaque branche du `match`.

```
fn main() {
    let pair = (2, -2);
    // TODO ^ Essayez de modifier les valeurs de `pair`.

    println!("Dites m'en plus à propos de: {:?}", pair);
    match pair {
        (x, y) if x == y => println!("Ils sont jumeaux!"),
        // La ^ condition if est un garde.
        (x, y) if x + y == 0 => println!("De l'antimatière, boom!"),
         (x, _) if x % 2 == 1 => println!("Le premier est étrange..."),
         _ => println!("Rien de spécial..."),
     }
}
```

Voir aussi

Les tuples.

7-5-3 - Assignation

Accéder indirectement à une variable rend impossible sa réutilisation sans la *réassigner*. `match` fournit le symbole `@` pour assigner des valeurs à des identificateurs :

```
// Une fonction `age` qui renvoie un entier de type `u32`.
fn age() -> u32 {
    15
}

fn main() {
    println!("Dis-moi quel type de personne es-tu.");

    match age() {
        0             => println!("Je ne suis pas encore né, je suppose."),
        // Nous aurions pu `match` 1 ... 12 directement mais il n'aurait pas
        // été possible de connaître l'âge de l'enfant
        // À la place, nous assignons la valeur à `n`
        // pour la séquence de 1 ... 12. L'âge peut désormais être affiché.
        n @ 1  ... 12 => println!("Je suis un enfant de {:?} ans!", n),
        n @ 13 ... 19 => println!("Je suis un adolescent de {:?} ans!", n),
        // Pas de limite. On renvoie le résultat.
        n             => println!("Je suis un adulte de {:?} ans!", n),
     }
}
```

Voir aussi

Les fonctions.

7-6 - if let

Pour certains cas, `match` peut être « lourd ». Par exemple :

```
// Crée une valeur optionnelle de type `Option<i32>`.
let optional = Some(7);

```

```
match optional {
    Some(i) => {
        println!("Ceci est une très longue chaîne de caractères contenant un
        `{:?}`", i);
        // ^ Deux niveaux d'indentations sont nécessaires alors
        // que nous aurions pu simplement déstructurer `i`.
    },
    _ => {},
    // ^ Nécessaire parce que `match` est exhaustif. Cette branche vous
    // paraît-elle utile?
};
```

if let est plus adapté à ce genre de cas et permet la création de plusieurs branches en cas d'erreur :

```
fn main() {
    // Toutes les variables sont de type `Option<i32>`.
    let number = Some(7);
    let letter: Option<i32> = None;
    let emoticon: Option<i32> = None;

    // L'ensemble `if let` se déroule de cette manière:
    // `if let` déstructure `number` et assigne sa valeur à `i` et exécute
    // le bloc (`{}`).
     if let Some(i) = number {
         println!("{:?} a été trouvé!", i);
     }

     // Si vous devez spécifier un cas d'erreur, utilisez un `else`:
     if let Some(i) = letter {
         println!("{:?} a été trouvé!", i);
     } else {
         // Déstructuration ratée. On exécute le `else`.
         println!("Aucun nombre n'a été trouvé.
         Cherchons une lettre!");
     };

     // Fournit une condition alternative.
     let i_like_letters = false;

     if let Some(i) = emoticon {
         println!("{:?} a été trouvé!", i);
     // Déstructuration ratée. Passe à une condition `else if` pour tester si
     // la condition alternative est vraie.
     } else if i_like_letters {
         println!("Aucun nombre n'a été trouvé.
         Cherchons une lettre!");
     } else {
         // La condition évaluée est fausse. Branche par défaut:
         println!("Je n'aime pas les lettres. Cherchons une emoticône :)!");
     };
}
```

Voir aussi

Les énumérations, **l'énumération Option** et **la RFC de if let**.

7-7 - while let

Ayant un fonctionnement similaire à if let, while let peut alléger la syntaxe de match lorsqu'il n'est pas nécessaire de passer par le *pattern matching*. Voici une séquence qui incrémente i :

```
// Crée une valeur optionnelle de type `Option<i32>`.
let mut optional = Some(0);

// On répète le test.
loop {
    match optional {
```

```
        // Si il est possible de déstructurer `optional`,
        // le bloc sera exécuté.
        Some(i) => {
            if i > 9 {
                println!("Plus grand que 9, on quitte!");
                optional = None;
            } else {
                println!("`i` est égal à `{:?}`. On réitère.", i);
                optional = Some(i + 1);
            }
            // ^ Nécessite trois niveaux d'indentations.
         },
         // On quitte la boucle si la déstructuration
         // a échoué:
         _ => { break; }
         // ^ Pourquoi cette instruction devrait être nécessaire ?
         // Il doit y avoir une solution plus adaptée!
     }
}
```

En utilisant while let, cela rend la séquence plus lisible :

```
fn main() {
    // Crée une valeur optionnelle de type `Option<i32>`.
    let mut optional = Some(0);

    // Fonctionnement: "`while let` déstructure `optional` pour assigner sa valeur
    // à Some(i) puis exécute le bloc (`{}`). Sinon, on sort de la boucle."
    while let Some(i) = optional {
        if i > 9 {
            println!("Plus grand que 9, on quitte!");
             optional = None;
         } else {
             println!("`i` est égal à `{:?}`. On réitère.", i);
             optional = Some(i + 1);
         }
         // Moins explicite, il n'est plus nécessaire de gérer
         // le cas où la déstructuration échoue.
     }
     // ^ `if let` permet d'ajouter des branches `else`/`else if`
     // optionnelles. `while let` ne le permet pas, en revanche.
}
```

Voir aussi

Les énumérations, **l'énumération Option** et **la RFC de while let**.

8 - Les fonctions

Les fonctions sont déclarées à l'aide du mot-clé fn. Leurs arguments sont typés, tout comme les variables, et, si la fonction renvoie une valeur, le type renvoyé doit être spécifié à la suite d'une flèche ->.

La dernière expression se trouvant dans le corps de la fonction sera utilisée pour inférer le type de renvoi. Il est également possible d'utiliser l'instruction return pour effectuer un renvoi prématuré dans la fonction (peut être utilisé dans les boucles et les structures conditionnelles).

Réécrivons les règles de FizzBuzz en utilisant les fonctions !

```
// Contrairement à C ou C++, l'ordre de déclaration des fonctions
// n'est pas important.
fn main() {
    // Nous pouvons appeler cette fonction ici et l'implémenter ailleurs.
    fizzbuzz_to(100);
}

```

```
// Une fonction qui renvoie une valeur booléenne.
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    // Comportement imprévisible, renvoi prématuré.
    if rhs == 0 {
        return false;
    }

    // C'est une expression, le mot-clé `return` n'est pas nécessaire ici.
    lhs % rhs == 0
}
// Les fonctions qui n'ont pas de type de renvoi défini renvoient par défaut
// un tuple vide `()`.
fn fizzbuzz(n: u32) -> () {
    if is_divisible_by(n, 15) {
        println!("fizzbuzz");
    } else if is_divisible_by(n, 3) {
        println!("fizz");
    } else if is_divisible_by(n, 5) {
        println!("buzz");
    } else {
        println!("{}", n);
    }
}

// Quand une fonction ne possède pas de type de renvoi, le tuple vide `()`
// peut être omis.
fn fizzbuzz_to(n: u32) /* type de renvoi omis */ {
    for n in 1..n + 1 {
        fizzbuzz(n);
    }
}
```

8-1 - Les méthodes

Les méthodes sont des fonctions rattachées à des structures et objets. Ces méthodes ont un accès aux données de l'objet ainsi qu'à ses autres méthodes par le biais du mot-clé self. Les méthodes sont déclarées dans un bloc impl.

```
struct Point {
    x: f64,
    y: f64,
}

// Toutes les méthodes de la structure `Point` sont implémentées ici.
impl Point {
    // Ceci est une méthode statique.
    // Les méthodes statiques ne sont pas dépendantes des instances.
    // Il est courant d'utiliser les méthodes statiques comme constructeurs.
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    // Une autre méthode statique possédant
    // deux paramètres.
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
}

struct Rectangle {
    p1: Point,
    p2: Point,
}

impl Rectangle {
    // Ceci est une méthode dépendante d'une instance (méthode d'instance).
    // `&self` est le sucre syntaxique de `self: &Self` où `Self` est le
    // type de l'objet qui appelle les méthodes. En l'occurrence
    // `Self` = `Rectangle`.
    fn area(&self) -> f64 {
```

```
        // `self` donne accès aux champs de la structure via la notation pointée.
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        // `abs` est une méthode renvoyant un réel de type f64 qui représente
        // la valeur absolue du primitif.
        ((x1 - x2) * (y1 - y2)).abs()
    }

    fn perimeter(&self) -> f64 {
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        2.0 * ((x1 - x2).abs() + (y1 - y2).abs())
    }

    // Cette méthode a besoin d'opérer sur une référence mutable
    // de l'instance courante `&mut self`.
    // La syntaxe non-raccourcie est `self: &mut Self`.
    fn translate(&mut self, x: f64, y: f64) {
        self.p1.x += x;
        self.p2.x += x;

        self.p1.y += y;
        self.p2.y += y;
    }
}

// `Pair` possède deux entiers alloués dans le tas.
struct Pair(Box<i32>, Box<i32>);

impl Pair {
    // Cette méthode "consomme" les ressources de l'instance courante.
    // `self` est un sucre syntaxique de `self: Self`.
    fn destroy(self) {
        // Déstructure `self` (i.e. récupère les champs désirés).
        let Pair(first, second) = self;

        println!("Destroying Pair({}, {})", first, second);

        // La mémoire occupée par `first` et `second` sera libérée
        // une fois que l'exécution de la méthode prendra fin.
    }
}

fn main() {
    let rectangle = Rectangle {
        // Les méthodes statiques sont appelées par le biais
        // d'une paire de deux points `::`.
        p1: Point::origin(),
        p2: Point::new(3.0, 4.0),
    };

    // Vous devez, en revanche, utiliser la notation pointée pour appeler
    // les méthodes d'instance.
    // Notez que le premier argument passé (implicitement) est `&self`
    // (i.e. `rectangle.perimeter()` === `Rectangle::perimeter(&rectangle)`).
    println!("Rectangle perimeter: {}", rectangle.perimeter());
    println!("Rectangle area: {}", rectangle.area());

    let mut square = Rectangle {
        p1: Point::origin(),
        p2: Point::new(1.0, 1.0),
    };

    // Erreur! `rectangle` est immuable alors que cette méthode
    // nécessite une référence mutable de l'objet.
    // rectangle.translate(1.0, 0.0);
    // TODO ^ Essayez de décommenter cette ligne.

    // C'est bon! Les objets mutables peuvent appeler les méthodes
    // "mutables".
```

```
    square.translate(1.0, 1.0);

    let pair = Pair(Box::new(1), Box::new(2));

    pair.destroy();

    // Erreur! L'appel de la méthode `destroy` a consommé
    // l'instance `pair`.
    // pair.destroy();
    // TODO ^ Essayez de décommenter cette ligne.
}
```

Définition : **valeur absolue**

8-2 - Les closures

Les closures en Rust, également appelées « lambdas », sont des fonctions qui peuvent capturer l'environnement qui les entoure. Par exemple, voici une closure qui capture la variable `x` :

```
|val| val + x
```

La syntaxe ainsi que les capacités des closures les rendent adéquates aux déclarations et utilisations à la volée. Appeler une closure se fait de la même manière qu'une fonction classique. En revanche, les types reçus en entrée (i.e. les types des paramètres passés) et le type de renvoi peuvent être **inférés** et les identificateurs des paramètres *doivent* être spécifiés.

D'autres caractéristiques spécifiques aux closures :

- L'utilisation du couple || plutôt que de () pour entourer les paramètres ;
- La délimitation {} du corps de la closure optionnelle pour une seule expression (sinon obligatoire) ;
- La capacité à capturer des variables appartenant au contexte dans lequel la closure est imbriquée.

```
fn main() {
    // Incrémentation avec les closures et fonctions.
    fn  function            (i: i32) -> i32 { i + 1 }

    // Les closures sont anonymes, ici nous assignons leurs références.
    // Les closures sont typées de la même manière qu'une fonction classique
    // mais le typage est optionnel. Chaque version (raccourcie et non-raccourcie)
    // est assignée à un identificateur approprié.
    let closure_annotated = |i: i32| -> i32 { i + 1 };
     let closure_inferred  = |i     |          i + 1  ;

     let i = 1;
     // Appelle la fonction et les closures.
     println!("function: {}", function(i));
     println!("closure_annotated: {}", closure_annotated(i));
     println!("closure_inferred: {}", closure_inferred(i));

     // Une closure qui ne prend aucun argument et renvoie un
     // entier de type `i32`.
     // Le type de renvoi est inféré.
     let one = || 1;
     println!("closure returning one: {}", one());

}
```

8-2-1 - Capture

Les closures sont naturellement flexibles et feront leur possible pour fonctionner sans typage explicite. Ceci permet à la capture de s'adapter au contexte : parfois en prenant possession des ressources, parfois seulement en les empruntant. Les closures peuvent capturer les variables :

- Par référence : &T ;
- Par référence mutable : &mut T ;
- Par valeur T.

Par défaut, elles privilégient la capture par référence s'il n'est pas nécessaire de prendre possession des ressources.

```
fn main() {
    use std::mem;

    let color = "green";

    // Une closure destinée à afficher la variable `color` qui emprunte
    // (`&`) `color` et stocke l'emprunt ainsi que la closure
    // dans la variable `print`. Elle (`color`) restera "empruntée"
    // jusqu'à ce que l'exécution de `print` prend fin.
    // La macro println! ne fait qu'emprunter les ressources, cela
    // n'impose pas de contraintes supplémentaires pour la closure.
    let print = || println!("`color`: {}", color);

    // On appelle la closure en empruntant `color`.
    print();
    print();

    let mut count = 0;

    // Une closure qui incrémente la variable `count`. Cette dernière
    // pourrait être exploitée par référence `&mut count` ou valeur
    // `count`, mais puisque `&mut count` est moins restrictif la capture
    // par référence mutable sera choisie.
    //
    // La variable `inc` est annotée comme `mut` car une référence mutable
    // `&mut` est stockée à l'intérieur.
    // Appeler la closure(du moins, dans ce contexte) modifie
    // son propre état et donc requiert un `mut`.

    let mut inc = || {
        count += 1;
        println!("`count`: {}", count);
    };

    // Appelle la closure.
    inc();
    inc();

    // let reborrow = &mut count;
    // ^ TODO: Essayez de décommenter cette ligne.

    // Un entier non-copiable.
    let movable = Box::new(3);

    // `mem::drop` prend possession de ses paramètres.
    // Un type pouvant être copié devrait être copié dans la closure,
    // laissant la ressource originale intacte. Un type qui ne peut pas
    // être copié doit être déplacé et donc `movable` appartiendra à la closure.
    let consume = || {
        println!("`movable`: {:?}", movable);
        mem::drop(movable);
    };

    // `consume` prend possession de la variable et ne peut donc être appelée qu'une seule fois.
    consume();
    // consume();
    // ^ TODO: Essayez de décommenter ce second appel.
}
```

Voir aussi

Box et **std::mem::drop**.

8-2-2 - Les closures passées en paramètres

Alors que Rust se charge de choisir, pour les closures, la manière de capturer les variables sans forcer le typage, lorsque c'est possible, cette ambiguïté n'est pas permise au sein des fonctions. Lorsqu'une closure est passée en paramètre à une fonction, le type de ses paramètres ainsi que celui de sa valeur de retour doivent être précisés en utilisant des traits. Dans l'ordre du plus restrictif au plus « laxiste » :

1. `Fn` : La closure capture par référence (`&T`) ;
2. `FnMut` : La closure capture par référence mutable (`&mut T`) ;
3. `FnOnce` : La closure capture par valeur (`T`).

En se fiant au contexte, le compilateur va capturer les variables en privilégiant le « régime » le moins restrictif possible.

Par exemple, prenez un paramètre typé avec le trait `FnOnce`. Cela signifie que la closure *peut* capturer ses variables par référence `&T`, référence mutable `&mut T`, ou valeur `T` mais le compilateur reste encore le seul juge quant au régime à adopter, en fonction du contexte.

C'est pourquoi si un transfert (`move`) est possible alors n'importe quel type d'emprunts devrait être possible, notez que l'inverse n'est pas vrai. Si le paramètre est typé `Fn` alors les captures par référence mutable `&mut T` ou par valeur `T` ne sont pas permises.

Dans l'exemple suivant, essayez de modifier le type de capture (i.e. `Fn`, `FnMut` et `FnOnce`) pour voir ce qu'il se passe :

```
// Une fonction qui prend une closure en paramètre et appelle cette dernière.
fn apply<F>(f: F) where
    // La closure ne prend rien et ne renvoie rien.
    F: FnOnce() {
    // ^ TODO: Essayez de remplacer ce trait par `Fn` ou `FnMut`.

    f();
}

// Une fonction qui prend une closure en paramètre et renvoie un entier
// de type `i32`.
fn apply_to_3<F>(f: F) -> i32 where
    // La closure prend en paramètre un `i32` et renvoie
    // un `i32`.
    F: Fn(i32) -> i32 {

    f(3)
}

fn main() {
    use std::mem;

    let greeting = "hello";
    // Un type qui ne peut pas être copié.
    // `to_owned` crée une ressource dont
    // l'assignation `farewell` sera responsable, à partir d'une ressource empruntée.
    let mut farewell = "goodbye".to_owned();

    // Capture deux variables: `greeting` par référence et
    // `farewell` par valeur.
    let diary = || {
        // `greeting` est capturé par référence: requiert `Fn`.
        println!("I said {}.", greeting);

        // Le fait de modifier `farewell` rend obligatoire
        // la capture par référence mutable, le compilateur choisira
        // donc `FnMut`.
        farewell.push_str("!!!");
        println!("Then I screamed {}.", farewell);
        println!("Now I can sleep. zzzzz");

```

```
        // Appeler manuellement la fonction `drop` nécessite
        // désormais de capturer par valeur `farewell`, le compilateur
        // choisira alors `FnOnce`.
        mem::drop(farewell);
    };

    // On appelle la fonction qui prend en paramètre la closure.
    apply(diary);

    // `double` satisfait les conditions du trait soumis à `apply_to_3`.
    let double = |x| 2 * x;

    println!("3 doubled: {}", apply_to_3(double));
}
```

Voir aussi

La fonction **std::mem::drop** et les traits **Fn**, **FnMut** et **FnOnce**.

8-2-3 - Les types anonymes

Les closures capturent succinctement les variables se trouvant dans les contextes qui les ont engendré. Cela a-t-il des conséquences ? Certainement. Nous remarquons qu'une fonction prête à recevoir une closure doit posséder un paramètre **générique** pour définir le « régime » de capture que la closure adoptera :

```
// `F` doit être générique.
fn apply<F>(f: F)
where
    F: FnOnce(),
{
    f();
}
```

Quand une closure est définie, le compilateur crée implicitement une *structure anonyme* pour stocker les variables capturées par la closure. Cette structure implémentera également l'un des traits rencontrés précédemment : `Fn`, `FnMut` ou `FnOnce`. Ce type anonyme est assigné à la variable stockée jusqu'à ce que la closure soit appelée.

Puisque le type créé implicitement est inconnu, son utilisation dans le corps d'une fonction nécessitera un paramètre générique. Cependant, un paramètre `<T>` dont le trait n'est pas précisé pourrait toujours être ambiguë et rejeté par le compilateur. Il est donc nécessaire de préciser quels services (i.e. `Fn`, `FnMut` ou `FnOnce`) il implémentera.

```
// `F` doit implémenter `Fn` pour une closure qui ne prend aucun
// argument et ne renvoie rien - exactement ce qui est nécessaire
// pour `print`.
fn apply<F>(f: F) where
    F: Fn() {
    f();
}

fn main() {
    let x = 7;

    // Capture la variable `x` dans une structure anonyme
    // et implémente `Fn` pour cette dernière. On stocke dans `print`.
    let print = || println!("{}", x);

    apply(print);
}
```

Voir aussi

Une analyse complète des closures, **Fn**, **FnMut** et **FnOnce**.

8-2-4 - Fonctions passées en paramètres

Les closures peuvent être soumises en entrée aux fonctions, mais vous pourriez vous demander si nous pouvons faire de même avec d'autres fonctions. C'est le cas ! Si vous déclarez une fonction qui prend une closure en paramètre alors n'importe quelle fonction implémentant les traits requis peut être passée en paramètre.

```
// On déclare une fonction qui prend l'argument générique `F`
// délimité par le trait `Fn` et appelle la fonction (ou closure).
fn call_me<F: Fn()>(f: F) {
    f();
}

// On déclare une fonction qui satisfait la délimitation (hériter de `Fn`).
fn function() {
    println!("I'm a function!");
}

fn main() {
    // On déclare une closure qui satisfait la délimitation (hériter de `Fn`).
    let closure = || println!("I'm a closure!");

    call_me(closure);
    call_me(function);
}
```

Voir aussi

Fn, **FnMut** et **FnOnce**.

8-2-5 - Renvoyer une closure

Les closures peuvent être passées en paramètre à une fonction, donc les renvoyer devrait être possible. Cependant, renvoyer un « type » de closure est problématique car, actuellement, Rust ne supporte le renvoi que de *types concrets* (i.e. non-génériques). Le type anonyme d'une closure est, par définition, inconnu donc le renvoi d'une closure ne peut être fait qu'en rendant son type concret.

Les traits destinés à valider le renvoi d'une closure sont quelque peu différents :

- Fn : pas de changements pour ce trait ;
- FnMut : pas de changements pour ce trait ;
- FnOnce : Différentes choses entrent en jeu ici, donc le type FnBox doit être utilisé à la place de FnOnce. Notez toutefois que FnBox est taggé **instable** et que des modifications pourraient être apportées dans le futur.

En dehors de cela, le mot-clé move doit être utilisé, indiquant que toutes les captures se feront par valeur pour la closure courante. Il est nécessaire d'utiliser move car aucune capture par référence ne pourrait être libérée aussitôt la fonction terminée, laissant des références invalides dans la closure.

```
fn create_fn() -> Box<Fn()> {
    let text = "Fn".to_owned();

    Box::new(move || println!("This is a: {}", text))
}

fn create_fnmut() -> Box<FnMut()> {
    let text = "FnMut".to_owned();

    Box::new(move || println!("This is a: {}", text))
}

fn main() {
```

```
    let fn_plain = create_fn();
    let mut fn_mut = create_fnmut();

    fn_plain();
    fn_mut();
}
```

Voir aussi

Box, **Fn**, **FnMut** et **la généricité**.

8-2-6 - Exemples de la bibliothèque standard

Cette section contient quelques exemples d'utilisation de closures avec des outils fournis par la bibliothèque standard.

8-2-6-1 - Iterator::any

Iterator::any est une fonction qui, lorsqu'un itérateur est passé en paramètre, renvoie true si au moins un élément satisfait le **prédicat**, autrement false. Voici sa signature :

```
pub trait Iterator {
    // Le type sur lequel on va itérer.
    type Item;

    // `any` prend en paramètre une référence mutable `&mut self` de
    // l'instance courante qui sera empruntée et modifiée, mais pas consommée
    // (possédée).
    fn any<F>(&mut self, f: F) -> bool
    where
        F: FnMut(Self::Item) -> bool,
     {
        true
     }
}
```

```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()`, pour les vecteurs, fournit la référence de chaque
    // élément `&i32`.
    println!("2 in vec1: {}", vec1.iter()     .any(|&x| x == 2));
    // `into_iter()`, pour les vecteurs, fournit la valeur de chaque élément `i32`.
    // L'itérateur est consommé.
     println!("2 in vec2: {}", vec2.into_iter().any(| x| x == 2));
     let array1 = [1, 2, 3];
     let array2 = [4, 5, 6];

     // `iter()` fournit la référence de chaque élément du tableau `&i32`.
     println!("2 in array1: {}", array1.iter()     .any(|&x| x == 2));
     // `into_iter()` fournit, exceptionnellement, la référence de chaque élément
     // du tableau `&i32` (le type i32 implémente les traits requis).
     println!("2 in array2: {}", array2.into_iter().any(|&x| x == 2));
}
```

Voir aussi

std::iter::Iterator::any.

8-2-6-2 - Iterator::find

Iterator::find est une fonction qui renvoie le premier élément correspondant au **prédicat**.

```
pub trait Iterator {
    // Le type sur lequel on va itérer.
    type Item;

    // `find` prend en paramètre une référence mutable de l'instance courante
    // `&mut self`. Elle sera donc empruntée et modifiée, mais pas consommée.
    fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
    where
        P: FnMut(&Self::Item) -> bool,
     {
         None
     }
}
```

```
fn main() {
    let vec1 = vec![1, 2, 3];
    let vec2 = vec![4, 5, 6];

    // `iter()` fournit la référence de chaque élément `&i32` du vecteur.
    let mut iter = vec1.iter();
    // `into_iter()` fournit la valeur de chaque élément du vecteur.
    let mut into_iter = vec2.into_iter();

     // Référence fournie par `iter`: `&&i32`. On déstructure la référence
     // de la référence pour obtenir l'entier `i32`.
     println!("Find 2 in vec1: {:?}", iter     .find(|&&x| x == 2));
     // Référence fournie par `into_iter`: `&i32`. On destructure la référence
     // pour obtenir l'entier `i32`.
     println!("Find 2 in vec2: {:?}", into_iter.find(| &x| x == 2));

     let array1 = [1, 2, 3];
     let array2 = [4, 5, 6];

     // `iter()` fournit la référence de chaque élément du tableau `&i32`.
     println!("Find 2 in array1: {:?}", array1.iter()     .find(|&&x| x == 2));
     // `into_iter()` fournit, exceptionnellement, la référence de chaque élément
     // du tableau `&i32` (le type i32 implémente les traits requis).
     println!("Find 2 in array2: {:?}", array2.into_iter().find(|&&x| x == 2));
}
```

Voir aussi

std::iter::Iterator::find.

8-3 - Les fonctions d'ordre supérieur

Rust supporte les **fonctions d'ordre supérieur** (HOF). Ces fonctions prennent en paramètre une ou plusieurs fonctions et renvoie une autre fonction. Ce sont les HOF ainsi que les « **itérateurs laxistes** » qui donnent cet aspect fonctionnel à Rust.

```
fn is_odd(n: u32) -> bool {
    n % 2 == 1
}

fn main() {
    println!("Find the sum of all the squared odd numbers under 1000");
    let upper = 1000;

    // Approche impérative.
     // On déclare un accumulateur.
     let mut acc = 0; // 0, 1, 2, ... ∞
     for n in 0.. {
         // Le carré du nombre.
         let n_squared = n * n;

         if n_squared >= upper {
             // On sort de la boucle si le carré de `n_squared`
```

```
            // dépasse la limite imposée par `upper`.
            break;
        } else if is_odd(n_squared) {
            // On accumule le carré de `n_squared` si c'est impair.
            acc += n_squared;
        }
    }
    println!("imperative style: {}", acc);

    // Approche fonctionnelle.
    let sum_of_squared_odd_numbers: u32 =
        (0..).map(|n| n * n)             // On calcule le carré de chaque nombre.
             .take_while(|&n| n <
 upper) // On vérifie que le carré se trouve toujours sous la limite de `upper`.
             .filter(|&n| is_odd(n))     // On récupère le nombre si il est impair.
             .fold(0, |sum, i| sum + i); // On accumule le carré du nombre impair.
    println!("functional style: {}", sum_of_squared_odd_numbers);
}
```

L'énumération **Option** et le trait **Iterator** implémentent leur lot d'HOF.

9 - Les modules

Rust fournit un puissant système de modules qui peut être utilisé pour hiérarchiser et diviser logiquement le code en plusieurs sous-modules et gérer la visibilité des ressources (publiques ou privées).

Un module est un ensemble d'éléments (e.g. ensemble de fonctions, de structures, de traits, de blocs « impl » et même d'autres modules).

9-1 - La visibilité

Par défaut, tout ce qui peut être contenu dans un module est privé. Nous pouvons remédier à cela en utilisant le mot-clé pub. Seuls les items publics d'un module peuvent être sollicités en dehors du contexte du module.

```
// Un module nommé `my`.
mod my {
    // Les items se trouvant dans le module sont privés, par défaut.
    fn private_function() {
        println!("called `my::private_function()`");
    }

    // Utilisez le mot-clé `pub` pour modifier la visibilité par défaut.
    pub fn function() {
        println!("called `my::function()`");
     }

     // Des items se trouvant dans le même module peuvent se solliciter
     // entre-eux, même lorsqu'ils sont privés.
     pub fn indirect_access() {
         print!("called `my::indirect_access()`, that\n> ");
         private_function();
     }

     // Les modules peuvent également être imbriqués.
     pub mod nested {
         pub fn function() {
             println!("called `my::nested::function()`");
         }

         #[allow(dead_code)]
         fn private_function() {
             println!("called `my::nested::private_function()`");
         }
     }
     // Les modules imbriqués suivent les mêmes règles vis-à-vis de la
```

```
    // visibilité.
    mod private_nested {
        #[allow(dead_code)]
        pub fn function() {
            println!("called `my::private_nested::function()`");
        }
    }
}

fn function() {
    println!("called `function()`");
}

fn main() {
    // Les noms des modules rattachés à une ressource peuvent être explicités
    // pour supprimer toute ambiguïté entre deux ressources possédant le même
    // nom.
    function();
    my::function();

    // Les items publics, y compris ceux qui se trouvent dans les modules
    // imbriqués, peuvent être sollicités en dehors du module parent.
    my::indirect_access();
    my::nested::function();


    // Les items privés d'un module ne peuvent pas être directement sollicités,
    // même si ils sont imbriqués dans un module public:

    // Erreur! `private_function` est privée.
    // my::private_function();
    // TODO ^ Essayez de décommenter cette ligne.

    // Erreur! `private_function` est privée.
    // my::nested::private_function();
    // TODO ^ Essayez de décommenter cette ligne.

    // Erreur! `private_nested` est un module privé.
    // my::private_nested::function();
    // TODO ^ Essayez de décommenter cette ligne.
}
```

9-2 - La visibilité des structures

Les structures disposent d'un niveau supplémentaire de visibilité dédié à leurs champs. Comme pour les autres ressources, les champs d'une structure sont privés par défaut, mais peuvent être rendus publics en utilisant, encore une fois, le mot-clé `pub`. La visibilité des champs ne s'applique, bien entendu, que lorsqu'une structure est sollicitée en dehors du module où elle a été déclarée et a pour but de masquer les données (**encapsulation**).

```
mod my {
    // Une structure publique avec un champ public générique de type `T`.
    pub struct WhiteBox<T> {
        pub contents: T,
    }

    // Une structure publique avec un champ privé générique de type `T`.
    #[allow(dead_code)]
    pub struct BlackBox<T> {
        contents: T,
    }

    impl<T> BlackBox<T> {
        // Constructeur public.
        pub fn new(contents: T) -> BlackBox<T> {
            BlackBox {
                contents: contents,
            }
        }
    }
```

```
}

fn main() {
    // Les structures publiques possédant des champs publics
    // peuvent être instanciées avec les séparateurs `{}`.
    let white_box = my::WhiteBox { contents: "public information" };

    // et leurs champs peuvent être sollicités normalement.
    println!("The white box contains: {}", white_box.contents);

    // Les structures publiques composées de champs privés ne peuvent pas être
    // instanciées de manière "classique" (i.e. en précisant le nom des champs).
    // Erreur! `BlackBox` possèdent des champs privés.
    // let black_box = my::BlackBox { contents: "classified information" };
    // TODO ^ Essayez de décommenter cette ligne.

    // En revanche, elles peuvent être créées avec un constructeur public.
    let _black_box = my::BlackBox::new("classified information");

    // Les champs privés d'une structure publique ne peuvent pas être
    // sollicités directement.
    // Erreur! Le champ `contents` est privé.
    // println!("The black box contains: {}", _black_box.contents);
    // TODO ^ Essayez de décommenter cette ligne.
}
```

Voir aussi

La généricité et **les méthodes**.

9-3 - La déclaration use

La déclaration `use` peut être utilisée pour assigner un chemin à *un nouveau nom*, pour y accéder plus rapidement.

```
// Assigne le chemin `deeply::nested::function` à l'identificateur
// `other_function`.
use deeply::nested::function as other_function;

fn function() {
    println!("called `function()`");
}

mod deeply {
    pub mod nested {
        pub fn function() {
            println!("called `deeply::nested::function()`")
        }
    }
}

fn main() {
    // Accès moins verbeux à `deeply::nested::function`.
    other_function();

    println!("Entering block");
    {
        // Ceci est équivalent à `use deeply::nested::function as function`.
        // La nouvelle assignation `function()` prendra le pas
        // sur `deeply::nested::function()`.
        use deeply::nested::function;
        function();

        // Les assignations `use` ne sont disponibles que dans le contexte
        // où elles ont vu le jour. Dans ce cas, où `function()` occulte
        // le chemin de base, ce "shadowing" n'est effectif que dans ce bloc.
        println!("Leaving block");
    }

```

```
    function();
}
```

9-4 - Les mot-clés super et self

Les mot-clés super et self peuvent être utilisés pour *lever l'ambiguïté sur la provenance d'une ressource* et éviter de réécrire leur chemin respectif(i.e. le chemin de la ressource).

```
fn function() {
    println!("called `function()`");
}

mod cool {
    pub fn function() {
        println!("called `cool::function()`");
    }
}

mod my {
    fn function() {
        println!("called `my::function()`");
    }

    mod cool {
        pub fn function() {
            println!("called `my::cool::function()`");
        }
    }

    pub fn indirect_call() {
        // Appelons toutes les fonctions nommées `fonction` depuis ce
        // contexte!
        print!("called `my::indirect_call()`, that\n> ");

        // Le mot-clé `self` fait référence au module courant (en l'occurrence
        // `my`). Appeler `self::function()` ou `function()` revient exactement
        // au même, puisqu'ils font tous deux référence à la même fonction.
        self::function();
        function();

        // Vous pouvez également utiliser `self` pour accéder à un autre module
        // imbriqué dans `my`.
        self::cool::function();

        // Le mot-clé `super` fait référence au contexte parent (en-dehors du
        // module `my`, pour cet exemple).
        super::function();

        // Si vous ne spécifiez aucun des deux mot-clés, le compilateur
        // comprendra que vous essayez d'utiliser une ressource se trouvant
        // dans le contexte de la crate.
        // On va donc assigner un nouvel identificateur à `cool::function()` se
        // trouvant dans le contexte de la crate.
        // Le contexte de la crate représente tout ce qui se trouve en-dehors des modules.
        {
            use cool::function as root_function;
            root_function();
        }
    }
}

fn main() {
    my::indirect_call();
}
```

9-5 - La hiérarchie des fichiers

Il est possible de transformer nos modules en un ensemble de fichiers et de répertoires. Recréons l'exemple utilisé pour illustrer le concept de **la visibilité** en un ensemble de fichiers :

```
$ tree .
.
|-- my
|   |-- inaccessible.rs
|   |-- mod.rs
|   `-- nested.rs
`-- split.rs
```

```
// Dans le fichier my/mod.rs
// De la même manière `mod inaccessible` et `mod nested` vont essayer de localiser
// les fichiers `nested.rs` et `inaccessible.rs` pour récupérer et ajouter leur contenu
// dans leur module respectif.
mod inaccessible;
pub mod nested;

pub fn function() {
    println!("called `my::function()`");
}

fn private_function() {
    println!("called `my::private_function()`");
}

pub fn indirect_access() {
    print!("called `my::indirect_access()`, that\n> ");

    private_function();
}
```

```
// Dans le fichier my/nested.rs
pub fn function() {
    println!("called `my::nested::function()`");
}

#[allow(dead_code)]
fn private_function() {
    println!("called `my::nested::private_function()`");
}
```

```
// Dans le fichier my/inaccessible.rs
#[allow(dead_code)]
pub fn public_function() {
    println!("called `my::inaccessible::public_function()`");
}
```

```
$ rustc split.rs && ./split
called `my::function()`
called `function()`
called `my::indirect_access()`, that
> called `my::private_function()`
called `my::nested::function()`
```

10 - Les crates

Une crate est une **unité de compilation** en Rust. Lorsque rustc some_file.rs est appelé, some_file.rs est considéré comme étant un «fichier paquet» (i.e. un fichier fédérant les autres). Si some_file.rs possède plusieurs modules en son sein, chacun d'entre eux verra son contenu fusionné dans ce paquet avant que la compilation n'ait lieu. Autrement dit, les modules ne sont pas compilés individuellement, mais en tant que paquet, en tant qu'ensemble de ressources.

Une crate peut être compilée en tant qu'exécutable ou bibliothèque. Par défaut, rustc produira un exécutable mais cela peut être modifié en passant le flag --crate-type au compilateur.

10-1 - Créer une bibliothèque

Commençons par créer une bibliothèque dont nous nous servirons ensuite pour l'importer dans une autre crate.

```
// Dans le fichier rary.rs
pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

    private_function();
}
```

```
$ rustc --crate-type=lib rary.rs
$ ls lib*
library.rlib
```

Les bibliothèques sont préfixées par la séquence « lib » et possèdent, par défaut, *le nom du fichier utilisé pour créer la crate* (en l'occurrence rary.rs). Ce comportement peut, bien entendu, être modifié en utilisant l'attribut **crate_name**.

10-2 - La déclaration extern crate

Pour importer une crate à cette nouvelle bibliothèque, il vous faudra utiliser la déclaration extern crate. Cette déclaration a aussi pour effet d'importer toutes les ressources sous un même module, possédant le même nom que la bibliothèque. Les règles régissant la visibilité des ressources s'appliquent également aux modules des bibliothèques importées.

```
// Dans le fichier executable.rs
// On importe la bibliothèque `library` et ses ressources
// sous un module nommé `rary`.
extern crate rary;

fn main() {
    rary::public_function();

    // Error! `private_function` est privée.
    // rary::private_function();

    rary::indirect_access();
}
```

```
# Où `library.rlib` est le chemin de la bibliothèque compilée, nous admetterons ici
# que la bibliothèque se trouve dans le répertoire courant.
$ rustc executable.rs --extern rary=library.rlib && ./executable
called rary's `public_function()`
called rary's `indirect_access()`, that
> called rary's `private_function()`
```

11 - Les attributs

Un attribut est une méta-donnée pouvant être appliqué à plusieurs sortes d'éléments (e.g. modules, crates). Ces méta-données peuvent être utilisées pour :

- Ajouter des **conditions à la compilation**;
- Établir **le nom, la version et le type** (i.e. exécutable ou bibliothèque) d'une crate ;
- Désactiver certains avertissements de **l'analyse**;
- Activer des fonctionnalités (e.g. macros, imports globaux) propres au compilateur ;
- Importer une bibliothèque d'un autre langage (i.e. **FFI**);
- Signaler des fonctions utilisées pour exécuter des tests unitaires ;
- Signaler des fonctions utilisées dans le cadre d'un benchmark.

Quand les attributs sont appliqués à une crate toute entière, leur syntaxe est la suivante #![crate_attribute]. Lorsqu'ils sont appliqués à un module ou un autre élément, la syntaxe est la suivante #[item_attribute](vous noterez la disparation du point d'exclamation).

Les attributs peuvent prendre des arguments sous différentes syntaxes :

- #[attribute = "value"] ;
- #[attribute(key = "value")] ;
- #[attribute(value)].

11-1 - L'avertissement dead_code

La compilateur fournit la **lint** dead_code qui vous avertira lorsqu'une instruction (ou une fonction) ne sera jamais exécutée. Un attribut peut être utilisé pour désactiver cette lint.

```
fn used_function() {}

// L'attribut `#[allow(dead_code)]` désactive la lint `dead_code`.
#[allow(dead_code)]
fn unused_function() {}

fn noisy_unused_function() {}
// FIXME ^ Ajoutez un attribut pour supprimer l'avertissement.

fn main() {
    used_function();
}
```

Gardez tout de même à l'esprit que, dans un programme destiné à être mis en production, il est préférable de supprimer le code mort. Ici, le code mort sera conservé simplement pour l'exemple.

11-2 - Méta-données relatives aux crates

L'attribut crate_type peut être utilisé pour renseigner au compilateur le type de la crate (i.e. exécutable ou bibliothèque(et quel type de bibliothèque)) et l'attribut crate_name est utilisé pour renseigner le nom de la crate.

```
// Dans le fichier lib.rs
// Cette crate est une bibliothèque.
#![crate_type = "lib"]
// Cette bibliothèque est nommée "rary".
#![crate_name = "rary"]

pub fn public_function() {
    println!("called rary's `public_function()`");
}

fn private_function() {
    println!("called rary's `private_function()`");
}

pub fn indirect_access() {
    print!("called rary's `indirect_access()`, that\n> ");

```

```
    private_function();
}
```

Lorsque l'attribut crate_type est utilisé vous n'avez, bien entendu, plus besoin de passer le flag --crate-type à rustc.

```
$ rustc lib.rs
$ ls lib*
library.rlib
```

11-3 - L'attribut cfg

La compilation conditionnelle est possible grâce à deux opérateurs :

1. L'attribut cfg : #[cfg(...)] ;
2. La macro cfg! : cfg!(...) en tant qu'expression booléenne.

Tous deux possèdent la même syntaxe :

```
// Cette fonction ne sera compilée que sur des distributions Linux.
#[cfg(target_os = "linux")]
fn are_you_on_linux() {
    println!("You are running linux!")
}

// Et cette fonction sera compilée seulement sur des plateformes qui ne sont
// *pas* des distributions Linux.
#[cfg(not(target_os = "linux"))]
fn are_you_on_linux() {
    println!("You are *not* running linux!")
}

fn main() {
    are_you_on_linux();

    println!("Are you sure?");
    if cfg!(target_os = "linux") {
        println!("Yes. It's definitely linux!");
    } else {
        println!("Yes. It's definitely *not* linux!");
    }
}
```

Voir aussi

La référence de l'attribut cfg, **std::cfg!** et **les macros**.

11-3-1 - Condition personnalisée

Certaines conditions (e.g. target_os) sont fournies par rustc. Il est toutefois possible de passer des conditions personnalisées à rustc en utilisant le flag --cfg.

```
// Dans le fichier custom.rs
#[cfg(some_condition)]
fn conditional_function() {
    println!("condition met!")
}

fn main() {
    conditional_function();
}
```

Sans le flag personnalisé :

```
$ rustc custom.rs && ./custom
No such file or directory (os error 2)
```

Avec le flag personnalisé :

```
$ rustc --cfg some_condition custom.rs && ./custom
condition met!
```

12 - La généricité

Comme son nom l'indique, cette section abordera les types et fonctionnalités génériques. La généricité peut être très utile pour réduire les répétitions au sein du code dans de nombreux cas, mais vous demandera, en échange, d'apporter quelques précisions supplémentaires à propos de la syntaxe. Notez également que rendre une ressource générique signifie que n'importe quelle ressource sera traitée de la même manière, il est nécessaire de savoir quels types de ressources peuvent être réellement traités, dans les cas où il est nécessaire de le spécifier.

La généricité est principalement utilisée pour rendre générique un, ou plusieurs, paramètre passé à une fonction. Par convention, un paramètre générique doit avoir un identificateur respectant la convention de nommage CamelCase et être déclaré entre un chevron ouvrant (`<`) et un chevron fermant(`>`) : `<Aaa, Bbb, ...>`, qui est souvent représenté par le paramètre `<T>`. En déclarant un paramètre générique de type `<T>`, on accepte de recevoir un, ou plusieurs, paramètre de ce type. Tout paramètre déclaré comme générique est générique, tout le reste est concret (non-générique).

Par exemple, voici une fonction générique nommée `foo` qui prend un paramètre de type `T` (de n'importe quel type, donc) :

```
fn foo<T>(p: T) { ... }
```

```
// A est un type concret.
struct A;

// Lorsque nous déclarons `Single`, la première occurrence de `A` n'est
// pas précédée du type générique `<A>`. Le type `Single` et `A` sont donc
// concrets.
struct Single(A);
//            ^ Voici la première occurrence du type `A`.

// En revanche, ici, `<T>` précède la première occurrence `T`, donc le type
// `SingleGen` est générique. Puisque le type `T` est générique, cela pourrait être
// "n'importe quoi", y compris le type concret `A` déclaré au début du fichier.
struct SingleGen<T>(T);

fn main() {
    // `Single` est un type concret et prend explicitement un paramètre
    // de type `A`.
    let _s = Single(A);

    // On créé une variable nommée `_char` de type `SingleGen<char>`
    // et on lui assigne la valeur `SingleGen('a')`.
    // Le type requis du paramètre passé pour cette instance de `SingleGen`
    // est spécifié, mais il peut être omis, exemple ---
    let _char: SingleGen<char> = SingleGen('a');

    // --->
    let _t    = SingleGen(A); // On passe une instance
                              // du type `A` définit en haut.
    let _i32  = SingleGen(6); // On passe un entier de type `i32`.
    let _char = SingleGen('a'); // On passe un `char`.
}
```

Voir aussi

Les structures

12-1 - Les fonctions

Les règles précédemment présentées s'appliquent également aux fonctions : un type T est générique lorsqu'il est précédé par la déclaration <T> (La lettre varie bien entendu selon le nom du type générique que vous donnez).

Lorsque vous utilisez des fonctions génériques il peut, parfois, être nécessaire d'expliciter le type des paramètres dans le cas où, par exemple, la fonction appelée possède un type de renvoi générique, ou encore si le compilateur ne dispose pas d'assez d'informations pour inférer le type des paramètres.

Une fonction dont le type des paramètres est explicité devrait ressembler à ceci : `fn::<A, B, ...>()`.

```
struct A;          // Le type `A` est concret.
struct S(A);       // Le type `S` est concret.
struct SGen<T>(T); // Le type `SGen` est générique.

// Toutes les fonctions qui suivront possèdent les paramètres
// qui leur sont passés et libèrent la mémoire aussitôt.

// On définit une fonction nommée `reg_fn` qui prend un argument `_s` de type
// `S`. Ce dernier n'est pas précédé d'un `<T>` (ou, en l'occurrence un `<S>`)
// donc le type n'est pas générique.
fn reg_fn(_s: S) {}

// On définit une fonction nommée `gen_spec_t` qui prend un argument `_s` de type
// `SGen<T>`.
// Le type d'argument imposé à la structure SGen<T> est précisé, mais puisque
// `A` n'est pas précédé par le type générique `<A>`, le type n'est pas générique.
fn gen_spec_t(_s: SGen<A>) {}
//          ^ il aurait fallu déclarer `A` comme générique: `<A>`.

// On définit une fonction nommée `gen_spec_i32` qui prend un argument `_s` de
// type `SGen<i32>`.
// Le type de paramètre supporté par la structure SGen a été spécifié, `i32`.
// Puisque `i32` n'est pas un type générique, cette fonction n'est pas générique non plus.
fn gen_spec_i32(_s: SGen<i32>) {}

// On définit une fonction nommée `generic` qui prend un paramètre `_s` de type
// `SGen<T>`.
// Puisque `SGen<T>` est précédé par le type générique `<T>`,
// cette fonction est donc générique.
fn generic<T>(_s: SGen<T>) {}

fn main() {
    // Appel des fonctions qui ne sont pas génériques.
    reg_fn(S(A));          // On passe en paramètre un type concret.
    gen_spec_t(SGen(A));   // Type `A` implicitement spécifié car la fonction
                           // n'accepte que la signature `SGen<A>`.
    gen_spec_i32(SGen(6)); // Type `i32` implicitement spécifié car la fonction
                           // n'accepte que la signature `SGen<i32>`.

    // Type du paramètre explicitement spécifié pour
    // la fonction `generic()`.
    // Le type peut être omis car le compilateur peut inférer
    // le type à partir du littéral.
    generic::<char>(SGen('a'));

    // Type du paramètre implicitement spécifié pour la fonction
    // `generic()` car le compilateur est capable d'inférer le type
    // à partir du littéral.
    generic(SGen('c'));
}
```

Voir aussi

Les fonctions, **les structures**.

12-2 - Implémentation générique

Tout comme les fonctions, les implémentations nécessitent quelques précisions pour être génériques.

```
struct S; // On déclare le type concret `S`.
struct GenericVal<T>(T); // On déclare le type générique `GenericVal`.

// Implémentation de GenericVal où nous précisons que cette méthode doit être
// implémentée uniquement pour le type `f32`.
impl GenericVal<f32> {
    fn say_hello_f32(&self) -> (){
        println!("I'm a float! :D");
    }
} // On spécifie `f32`
impl GenericVal<S> {
    fn say_hello_s(&self) -> (){
        println!("I'm a S object! :D");
    }
} // On spécifie le type `S` pour les mêmes raisons qu'au-dessus.
// `<T>` doit précéder le type pour le rendre générique.
impl <T> GenericVal<T> {
    fn say_hello(&self) -> (){
        println!("I'm a generic object! :D");
    }
}

struct Val {
    val: f64
}

struct GenVal<T>{
    gen_val: T
}

// Implémentation de Val.
impl Val {
    fn value(&self) -> &f64 { &self.val }
}

// Implémentation de GenVal pour le type générique `T`.
impl <T> GenVal<T> {
    fn value(&self) -> &T { &self.gen_val }
}

fn main() {
    let x = Val { val: 3.0 };
    let y = GenVal { gen_val: 3i32 };

    println!("{}, {}", x.value(), y.value());
    GenericVal(1.0).say_hello_f32();
    GenericVal(S).say_hello_s();
    GenericVal("prout").say_hello();
}
```

Voir aussi

Renvoi de références, **impl**, **les structures**.

12-3 - Les traits

Bien entendu, les traits peuvent également être génériques. Dans cette section, nous allons en créer un qui ré-implémente le trait Drop qui proposera une méthode générique qui aura pour fonction de libérer l'instance qui l'appelle ainsi qu'un paramètre passé.

```
// Types non-copiables.
struct Empty;
struct Null;

// Un trait générique qui reçoit un type générique `T`.
trait DoubleDrop<T> {
    // On déclare une méthode qui sera implémentée par la structure
  // appelante (caller) et prendra un paramètre `T` en entrée mais n'en fera rien (juste le libérer).
    fn double_drop(self, _: T);
}

// On implémente `DoubleDrop<T>` pour n'importe quel paramètre `T` et
// n'importe quelle structure appelante (`U`).
impl<T, U> DoubleDrop<T> for U {
    // Cette méthode prend "possession" des deux paramètres (`U` et `T`),
    // et sont donc tous deux libérés.
    fn double_drop(self, _: T) {}
}

fn main() {
    let empty = Empty;
    let null  = Null;

    // `empty` et `null` sont désalloués.
    empty.double_drop(null);

    // empty;
    // null;
    // ^ TODO: Essayez de décommenter ces lignes.
}
```

Voir aussi

La documentation du trait Drop, **le chapitre sur les structures** et **les traits**.

12-4 - Les restrictions

Lorsque nous travaillons avec la généricité, il est courant d'assigner une « restriction » à un type générique pour spécifier quel trait il doit implémenter. Dans l'exemple suivant, nous utilisons le trait Display pour afficher quelque chose en console, il est alors assigné au type T. Autrement dit, T doit implémenter Display.

```
// On déclare une fonction nommée `printer` qui prend, en entrée,
// un type générique `T` qui doit implémenter le trait `Display`.
fn printer<T: Display>(t: T) {
    println!("{}", t);
}
```

Les ressources passées en paramètre sont toutes soumises à ces restrictions et doivent forcément remplir les conditions :

```
struct S<T: Display>(T);

// Erreur! `Vec<T>` n'implémente pas le trait `Display`.
let s = S(vec![1]);
```

Les instances des types génériques peuvent également accéder aux méthodes appartenant au(x) trait(s) présent(s) dans les restrictions du type. Par exemple :

```
// Un trait qui implémente le marqueur `{:?}`.
use std::fmt::Debug;

trait HasArea {
    fn area(&self) -> f64;
}
```

```

impl HasArea for Rectangle {
    fn area(&self) -> f64 { self.length * self.height }
}

#[derive(Debug)]
struct Rectangle { length: f64, height: f64 }
#[allow(dead_code)]
struct Triangle  { length: f64, height: f64 }

// Le type générique `T` doit implémenter le trait `Debug`.
// Qu'importe le type de `T`, cela fonctionnera.
fn print_debug<T: Debug>(t: &T) {
    println!("{:?}", t);
}

// `T` doit implémenter le trait `HasArea`. N'importe quelle
// structure remplissant les conditions d'entrée peut accéder
// à la méthode `area` du trait `HasArea`.
fn area<T: HasArea>(t: &T) -> f64 { t.area() }

fn main() {
    let rectangle = Rectangle { length: 3.0, height: 4.0 };
    let _triangle = Triangle  { length: 3.0, height: 4.0 };

    print_debug(&rectangle);
    println!("Area: {}", area(&rectangle));

    // print_debug(&_triangle);
    // println!("Area: {}", area(&_triangle));
    // ^ TODO: Essayez de décommenter ces lignes.
    // | Erreur: N'implémente pas l'un de ces traits: `Debug` ou `HasArea`.
}
```

Les conditions d'entrée pour les paramètres génériques peuvent également être spécifiées en utilisant le mot-clé **where**, les rendant plus explicites, plus lisibles.

Voir aussi

Les traits dédiés à l'affichage et le formatage **std::fmt**, **les structures** et **les traits**.

12-4-1 - Exemple d'utilisation : Traits sans services

Puisqu'il est possible d'imposer des conditions aux types génériques grâce aux traits, même si ces derniers ne possèdent aucune fonctionnalité (i.e. aucun service, aucune méthode), il est toujours possible de vous en servir comme simple « filtre ». Eq et Ord font partie de ces traits « vides » fournis par la bibliothèque standard.

```
struct Cardinal;
struct BlueJay;
struct Turkey;

trait Red {}
trait Blue {}

impl Red for Cardinal {}
impl Blue for BlueJay {}

// Ces fonctions ne prendront en entrée que des ressources
// ayant implémenté les traits `Red` ou `Blue`.
// Le fait que ces derniers soient vides n'a que peu d'importance.
fn red<T: Red>(_: &T)   -> &'static str { "red" }
fn blue<T: Blue>(_: &T) -> &'static str { "blue" }

fn main() {
    let cardinal = Cardinal;
    let blue_jay = BlueJay;
    let _turkey   = Turkey;
```

```
    // La fonction `red()` ne fonctionnera pas sur une instance
    // de la structure `BlueJay`, et vice versa, à cause des
    // restrictions imposées par les fonctions (i.e. `red()` et `blue()`).
    println!("A cardinal is {}", red(&cardinal));
    println!("A blue jay is {}", blue(&blue_jay));
    //println!("A turkey is {}", red(&_turkey));
    // ^ TODO: Essayez de décommenter cette ligne.
}
```

Voir aussi

La documentation du trait **Eq**, la documentation du trait **Ord** et **les traits**.

12-5 - Restrictions multiples

Il est possible d'additionner les conditions grâce à l'opérateur +. Tout comme les types concrets, les types génériques sont séparés par une virgule ,.

```
use std::fmt::{Debug, Display};

fn compare_prints<T: Debug + Display>(t: &T) {
    println!("Debug: `{:?}`", t);
    println!("Display: `{}`", t);
}

fn compare_types<T: Debug, U: Debug>(t: &T, u: &U) {
    println!("t: `{:?}", t);
    println!("u: `{:?}", u);
}

fn main() {
    let string = "words";
    let array = [1, 2, 3];
    let vec = vec![1, 2, 3];

    compare_prints(&string);
    // compare_prints(&array);
    // TODO ^ Essayez de décommenter cette ligne.

    compare_types(&array, &vec);
}
```

Voir aussi

std::fmt, **les traits**.

12-6 - La clause where

Une restriction peut également être explicitée par la clause where. Cette dernière se trouvera alors avant l'accolade ouvrante ({) plutôt qu'à la déclaration du type(e.g. <A: Display, B: Debug, ...>). Avec where, vous pouvez également ajouter arbitrairement d'autres types en plus de spécifier les traits à implémenter pour les types génériques.

where peut être utile dans plusieurs cas :

- Lorsque vous ajoutez des restrictions aux types génériques, facilitant la lecture :

```
impl <A: TraitB + TraitC, D: TraitE + TraitF> MyTrait<A, D> for YourType {}

// Les restrictions sont explicitées
// par la condition `where`.
impl <A, D> MyTrait<A, D> for YourType where
```

```
    A: TraitB + TraitC,
    D: TraitE + TraitF {}
```

- Lorsque d'autres types sont ajoutés aux restrictions :

```
use std::fmt::{Debug, Formatter, Result};

trait PrintInOption {
    fn print_in_option(self);
}

// Parce que nous pourrions modifier la restriction sans spécifier le
// conteneur `T: Debug` ou adopter une autre approche,
// il est nécessaire d'utiliser la condition `where`:
impl<T> PrintInOption for T where
    Option<T>: Debug{
    // Nous spécifions la restriction de type `Option<T>: Debug` parce que
    // c'est ce que nous souhaitons afficher. Faire autrement pourrait nous
    // induire en erreur quant au type de restriction à spécifier.
    fn print_in_option(self) {
        println!("{:?}", Some(self));
    }
}

fn main() {
    let vec = vec![1, 2, 3];
    vec.print_in_option();
}
```

Voir aussi

RFC pour la condition/clause where, **les structures**, **les traits**.

12-7 - Les éléments associés

Le concept « d'éléments associés » est un ensemble de règles appliquées sur différents types **d'éléments**. C'est une extension des traits génériques, leur permettant de définir de nouveaux « éléments » en leur sein ; En l'occurrence, ces derniers sont nommés « types associés » et proposent une approche moins fastidieuse pour l'utilisation des patterns (e.g. déclaration des types) lorsqu'un trait reçoit des paramètres génériques en entrée.

Voir aussi

La **RFC des éléments associés**.

12-7-1 - Le problème

Un trait qui possède des types génériques en entrée doit respecter certaines règles : Les utilisateurs du trait doivent spécifier tous les types de données génériques supportés par le conteneur.

Dans l'exemple ci-dessous, le trait Contains permet l'utilisation des types génériques A et B. Le trait est alors implémenté pour le type Container en spécifiant le type i32 pour les deux types génériques (i.e. A et B). Ils peuvent donc être soumis à la fonction fn difference().

Puisque le type Contains est générique, nous sommes obligés de déclarer tous les types génériques supportés par Contains pour la fonction difference(). En pratique, nous opterions pour une approche nous permettant d'inférer les types génériques supportés par Contains à partir de l'entrée C. C'est ce que nous verrons dans la section suivante, car les types associés autorisent cette approche.

```
struct Container(i32, i32);

```

```
// Nous déclarons un trait qui vérifie si deux items sont contenus
// par le conteneur.
// Le conteneur pourra également fournir la première ou dernière
// valeur.
trait Contains<A, B> {
    fn contains(&self, &A, &B) -> bool;
    fn first(&self) -> i32;
    fn last(&self) -> i32;
}

impl Contains<i32, i32> for Container {
    // Renvoie `true` si les nombres stockés sont égaux.
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }

    // On récupère la première valeur.
    fn first(&self) -> i32 { self.0 }

    // On récupère la dernière valeur.
    fn last(&self) -> i32 { self.1 }
}

// `C` contient `A` et `B`. Les déclarer une nouvelle fois dans les paramètres
// génériques de la fonction est fastidieux.
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> {
    container.last() - container.first()
}

fn main() {
    let number_1 = 3;
    let number_2 = 10;

    let container = Container(number_1, number_2);

    println!("Does container contain {} and {}: {}",
        &number_1, &number_2,
        container.contains(&number_1, &number_2));
    println!("First number: {}", container.first());
    println!("Last number: {}", container.last());

    println!("The difference is: {}", difference(&container));
}
```

Voir aussi

Les structures.

12-7-2 - Les types associés

L'utilisation des « types associés » améliore la lisibilité du code en assignant les types génériques, au sein du trait, comme « types de sortie ». La syntaxe pour les déclarer est la suivante :

```
// `A` et `B` sont déclarés au sein du trait par le biais du mot-clé `type`.
// Notez toutefois que `type`, dans ce contexte, n'a pas la même fonction
// que le `type` utilisé pour créer des alias.
trait Contains {
    type A;
    type B;

    // La syntaxe a été modifiée pour faire référence aux types génériques.
    fn contains(&self, &Self::A, &Self::B) -> bool;
}
```

Notez que les fonctions qui utilisent le trait `Contains` n'ont plus du tout besoin de déclarer les types génériques `A` et `B` :

```
// Sans les types associés.
fn difference<A, B, C>(container: &C) -> i32 where
    C: Contains<A, B> { ... }

// Avec les types associés.
fn difference<C: Contains>(container: &C) -> i32 { ... }
```

Éditons l'exemple de la section précédente en utilisant les types associés :

```
struct Container(i32, i32);

// Nous déclarons un trait qui vérifie si deux items sont stockés
// dans le conteneur.
// Le conteneur pourra également fournir la première ou dernière valeur.
trait Contains {
    // Nous définissons une bonne fois pour tous les types génériques
    // que les méthodes/fonctions pourront utiliser.
    type A;
    type B;

    fn contains(&self, &Self::A, &Self::B) -> bool;
    fn first(&self) -> i32;
    fn last(&self) -> i32;
}

impl Contains for Container {
    // On précise le type de `A` et `B`. Si le type d'entrée est
    // `Container(i32, i32)`, les types de sorties seront alors
    // typés tous les deux `i32`.
    type A = i32; // typé i32
    type B = i32; // typé i32

    // `&Self::A` et `&Self::B` sont également valides ici.
    // Essayez de remplacer la référence number_1(`&i32`) par `&Self::A`,
    // et la référence number_2(`&i32) par `&Self::B`!
    fn contains(&self, number_1: &i32, number_2: &i32) -> bool {
        (&self.0 == number_1) && (&self.1 == number_2)
    }
    // Récupère la première valeur.
    fn first(&self) -> i32 { self.0 }

    // Récupère la dernière valeur.
    fn last(&self) -> i32 { self.1 }
}

fn difference<C: Contains>(container: &C) -> i32 {
    container.last() - container.first()
}

fn main() {
    let number_1 = 3;
    let number_2 = 10;

    let container = Container(number_1, number_2);

    println!("Does container contain {} and {}: {}",
        &number_1, &number_2,
        container.contains(&number_1, &number_2));
    println!("First number: {}", container.first());
    println!("Last number: {}", container.last());

    println!("The difference is: {}", difference(&container));
}
```

12-8 - Les paramètres fantômes

Un type de paramètre fantôme n'est pas utilisé à l'exécution, mais est vérifié statiquement (et seulement) au moment de la compilation.

Les types de données peuvent utiliser des types de paramètres génériques supplémentaires pour agir en tant que « marqueurs » ou pour effectuer une vérification du/des type(s) au moment de la compilation. Ces paramètres « supplémentaires » ne stockent aucune ressource et sont inactifs à l'exécution.

Dans l'exemple ci-dessous, nous présentons la structure **std::marker::PhantomData** avec le concept de « type de paramètre fantôme » pour créer des tuples contenant différents types de données.

```
use std::marker::PhantomData;

// Un tuple qui prend un type générique `A` et un paramètre fantôme `B`.
#[derive(PartialEq)] // Permet de tester l'égalité entre les instances du type.
struct PhantomTuple<A, B>(A,PhantomData<B>);

// Un tuple qui prend un type générique `A` et un paramètre fantôme `B`.
#[derive(PartialEq)] // Permet de tester l'égalité entre les instances du type.
struct PhantomStruct<A, B> { first: A, phantom: PhantomData<B> }

// Note: De la mémoire sera allouée pour le type générique `A`, mais pas pour `B`.
//       En revanche, `B` ne pourra pas être utilisé à l'exécution.

fn main() {
    // Ici, les types `f32` et `f64` sont des paramètres fantômes.
    // Types spécifiés pour PhantomTuple: `<char, f32>`.
    let _tuple1: PhantomTuple<char, f32> = PhantomTuple('Q', PhantomData);
    // Types spécifiés pour PhantomTuple: `<char, f64>`.
    let _tuple2: PhantomTuple<char, f64> = PhantomTuple('Q', PhantomData);

    // Types spécifiés pour PhantomStruct: `<char, f32>`.
    let _struct1: PhantomStruct<char, f32> = PhantomStruct {
        first: 'Q',
        phantom: PhantomData,
    };
    // Types spécifiés pour PhantomStruct: `<char, f64>`.
    let _struct2: PhantomStruct<char, f64> = PhantomStruct {
        first: 'Q',
        phantom: PhantomData,
    };

    // Erreur! Les deux ressources ne peuvent pas être comparées:
    // println!("_tuple1 == _tuple2 yields: {}",
    //           _tuple1 == _tuple2);

    // Erreur! Les deux ressources ne peuvent pas être comparées:
    // println!("_struct1 == _struct2 yields: {}",
    //           _struct1 == _struct2);
}
```

Voir aussi

L'attribut Derive, **les structures** et **les tuples**.

12-8-1 - Exemple d'utilisation Petite précision

Nous allons créer une méthode chargée de calculer dans deux unités de mesure différentes (**le pied** et **le millimètre**) et nous implémenterons le trait Add avec un type générique fantôme. Voici l'implémentation du trait Add :

```
// Cette implémentation devrait imposer: `Self + RHS = Output`
// où `Self` est la valeur par défaut de `RHS` si elle n'est pas spécifiée
// dans l'implémentation.

pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}

```

```
// `Output` doit être de type `T<U>` pour que `T<U> + T<U> = T<U>`.
impl<U> Add for T<U> {
    type Output = T<U>;
    ...
}
```

Voici l'implémentation complète :

```
use std::ops::Add;
use std::marker::PhantomData;

/// On créé des énumérations vides pour déclarer le type des
// unités de mesures.
#[derive(Debug, Clone, Copy)]
enum Inch {}
#[derive(Debug, Clone, Copy)]
enum Mm {}

/// `Length` est une structure prenant un type générique fantôme `Unit`,
/// `f64` implémente déjà les traits `Clone` et `Copy`.
#[derive(Debug, Clone, Copy)]
struct Length<Unit>(f64, PhantomData<Unit>);

/// Le trait `Add` définit le comportement de l'opérateur `+`.
impl<Unit> Add for Length<Unit> {
     type Output = Length<Unit>;

    // La méthode add() renvoie une nouvelle instance de la
    // structure `Length` contenant la somme.
    fn add(self, rhs: Length<Unit>) -> Length<Unit> {
        // L'opérateur `+` appelle l'implémentation
        // du trait `Add` pour le type `f64`.
        Length(self.0 + rhs.0, PhantomData)
    }
}

fn main() {
    // On initialise `one_foot` pour avoir un type générique fantôme `Inch`.
    // Ce "fantôme" sert de marqueur et classe cette instance dans
    // l'unité de mesure "Pied".
    let one_foot:  Length<Inch> = Length(12.0, PhantomData);
    // On initialise `one_meter` pour avoir un type générique fantôme `Mm`.
    // Ce "fantôme" sert de marqueur et classe cette instance dans
    // l'unité de mesure "Millimètre".
    let one_meter: Length<Mm>   = Length(1000.0, PhantomData);

    // L'opérateur `+` appelle la méthode `add()` que nous avons
    // précédemment implémentée pour `Length<Unit>`.
    // Maintenant que `Length` implémente le trait `Copy`, `add()` ne
    // prend pas possession (ne consomme pas) `one_foot` et `one_meter` mais
    // les copie dans `self` et `rhs`.
    let two_feet = one_foot + one_foot;
    let two_meters = one_meter + one_meter;

    // L'addition fonctionne.
    println!("one foot + one_foot = {:?} in", two_feet.0);
    println!("one meter + one_meter = {:?} mm", two_meters.0);

    // Les opérations illogiques échouent comme prévu:
    // Erreur à la compilation: mismatched type.
    // let one_feter = one_foot + one_meter;
}
```

Voir aussi

Le système d'emprunts, **les restrictions**, **les énumérations**, **impl et self**, **la surcharge des opérateurs**, **le pattern ref**, **les traits** et **les tuples**.

13 - Les contextes

Les contextes jouent un rôle important dans le fonctionnement du systèmes de propriété (ownership), d'emprunts (borrowing) et de durées de vie (lifetime). Ils témoignent de la validité (ou non) d'un emprunt, indiquent au compilateur lorsqu'une ressource peut être libérée et lorsqu'une variable est créée ou détruite.

13-1 - Le RAII

En Rust, les variables ne stockent pas seulement leurs données dans la pile : Elles sont responsables de leurs ressources (e.g. le conteneur Box<T> possède, alloue de la mémoire dans le tas). Rust imposant l'approche du RAII, lorsqu'un objet sort du contexte, son destructeur est appelé et les ressources, possédées par l'objet, sont libérées.

Ce fonctionnement prévient les problèmes de fuites mémoire et nous dispense donc de gérer manuellement la mémoire. Voici un exemple :

```
// Dans le fichier raii.rs
fn create_box() {
    // Allocation d'un entier dans le tas.
    let _box1 = Box::new(3i32);

    // `_box1` est détruit ici, et la mémoire est libérée.
}

fn main() {
    // Allocation d'un entier dans le tas.
    let _box2 = Box::new(5i32);

    // Contexte imbriqué:
    {
        // Allocation d'un entier dans le tas.
        let _box3 = Box::new(4i32);

        // `_box3` est détruit ici, et la mémoire est libérée.
    }

    // On créé ici un grand nombre de "box" juste pour l'exemple.
    // Il n'y a pas besoin de libérer manuellement la mémoire !
    for _ in 0u32..1_000 {
        create_box();
    }

    // `_box2` est détruit ici, et la mémoire est libérée.
}
```

Bien entendu, vous pouvez vérifier par vous-même si des fuites sont présentes en utilisant **valgrind** :

```
$ rustc raii.rs && valgrind ./raii
==26873== Memcheck, a memory error detector
==26873== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==26873== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==26873== Command: ./raii
==26873==
==26873==
==26873== HEAP SUMMARY:
==26873==     in use at exit: 0 bytes in 0 blocks
==26873==   total heap usage: 1,013 allocs, 1,013 frees, 8,696 bytes allocated
==26873==
==26873== All heap blocks were freed -- no leaks are possible
==26873==
==26873== For counts of detected and suppressed errors, rerun with: -v
==26873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Voir aussi

Box.

13-2 - Ownership et transferts

Parce que les variables sont responsables de la libération de leurs ressources, **les ressources ne peuvent avoir qu'un seul propriétaire/responsable**.

Cette règle évite également au développeur de libérer plus d'une fois une ressource. Notez toutefois que toutes les variables ne possèdent pas leurs propres ressources (e.g. **les références**).

Lorsque nous assignons quelque chose à une variable (`let x = y`) ou passons un (ou des) argument à une fonction par valeur (`foo(x)`), l'ownership des ressources est transféré. Dans le jargon, cette action est nommée « `move` » (transfert).

Après avoir transféré une ressource, l'ancien propriétaire ne peut plus être utilisé. Cela prévient la création de « dangling pointers » (i.e. des pointeurs sur une ressource qui n'est plus valide).

```rust
// Cette fonction prend possession d'un entier alloué dans le tas.
fn destroy_box(c: Box<i32>) {
    println!("Destroying a box that contains {}", c);

    // `c` est détruit et la mémoire va être libérée.
}

fn main() {
    // Entier alloué dans la pile.
     let x = 5u32;

     // On copie `x` dans `y` - aucune ressource n'a été transféré
     // (l'ownership n'a pas été transféré).
     let y = x;

     // Les deux valeurs peuvent être utilisées indépendamment.
     println!("x is {}, and y is {}", x, y);

     // `a` est un pointeur sur un entier alloué dans le tas.
     let a = Box::new(5i32);

     println!("a contains: {}", a);

     // On transfert `a` dans `b`.
     let b = a;
     // L'adresse du pointeur `a` est copié (et non la donnée) dans `b`.
     // `a` et `b` sont désormais des pointeurs sur la même donnée allouée dans le
     // tas, mais `b` la possède, désormais.

     // Erreur! `a` ne peut plus accéder à la donnée car il ne possède plus
     // le bloc mémoire.
     // println!("a contains: {}", a);
     // TODO ^ Essayez de décommenter cette ligne.

     // Cette fonction prend possession de la mémoire allouée dans le tas
     // à partir de `b`.
     destroy_box(b);

     // Puisque la mémoire allouée a été libérée à partir d'ici,
     // cette action consisterait à déréférencer de la mémoire libérée,
     // mais cela est interdit par le compilateur.
     // Erreur! `b` ne peut plus accéder à la donnée car il ne possède plus
     // le bloc mémoire.
     // println!("b contains: {}", b);
     // TODO ^ Essayez de décommenter cette ligne.
}
```

13-2-1 - Mutabilité

La mutabilité d'une donnée peut être altérée lorsque l'ownership est transféré.

```
fn main() {
    let immutable_box = Box::new(5u32);

    println!("immutable_box contains {}", immutable_box);

    // Erreur: `immutable_box` ne peut pas être déréférencé.
    // *immutable_box = 4;

    // On créé une copie mutable de `immutable_box`.
     let mut mutable_box = immutable_box;

     println!("mutable_box contains {}", mutable_box);

     // On modifie le contenu de la box.
     *mutable_box = 4;

     println!("mutable_box now contains {}", mutable_box);
}
```

13-3 - Système d'emprunts

Très souvent, nous souhaiterions accéder à une ressource sans en prendre possession. Pour ce faire, Rust utilise un *système d'emprunts*. Plutôt que de passer un objet *par valeur* (`T`), il peut être passé par référence (`&T`).

Le compilateur garantit (grâce au vérificateur d'emprunts) que les références sont toujours valides. Tant qu'une référence de l'objet existe, il ne sera pas détruit.

```
// Cette fonction prend possession d'une box et la détruit.
fn eat_box_i32(boxed_i32: Box<i32>) {
    println!("Destroying box that contains {}", boxed_i32);
}

// Cette fonction emprunte un entier signé
// codé sur 32 bits.
fn borrow_i32(borrowed_i32: &i32) {
    println!("This int is: {}", borrowed_i32);
}

fn main() {
     // On créé entier alloué dans le tas
     // et un autre alloué dans la pile.
     let boxed_i32 = Box::new(5_i32);
     let stacked_i32 = 6_i32;

     // `borrow_i32()` emprunte le contenu de la box.
     // La fonction ne prend pas possession des ressources,
     // donc le contenu peut être de nouveau emprunté.
     borrow_i32(&boxed_i32);
     borrow_i32(&stacked_i32);

     {
         // Récupère une référence de l'entier contenu dans la box.
         let _ref_to_i32: &i32 = &boxed_i32;

         // Erreur!
         // Vous ne pouvez pas détruire `boxed_i32` tant que la valeur
         // qu'il contient est empruntée.
         // eat_box_i32(boxed_i32);
         // FIXME ^ Essayez de décommenter cette ligne.

         // La durée de vie de `_ref_to_i32` prend fin ici,
         // l'entier n'est donc plus emprunté.
```

```
    }

    // `boxed_i32` peut désormais être possédé par `eat_box()` et peut donc
    // être détruit.
    eat_box_i32(boxed_i32);
}
```

13-3-1 - Mutabilité

Une ressource mutable peut être modifiée, tout en étant empruntée, en utilisant &mut T. Nous passons alors une référence mutable de la ressource, donnant un accès en lecture et en écriture à l'emprunteur. En revanche, une référence dont la mutabilité n'est pas précisée (i.e. que le mot-clé mut n'est pas présent) empêche l'emprunteur d'accéder en écriture à la ressource.

```
#[allow(dead_code)]
#[derive(Clone, Copy)]
struct Book {
    // `&'static str` est une référence d'une chaîne de caractères allouée
    // dans un bloc mémoire qui ne peut être accédé qu'en lecture.
    author: &'static str,
    title: &'static str,
    year: u32,
}

// Cette fonction prend une référence d'une instance
// de la structure `Book` en paramètre.
fn borrow_book(book: &Book) {
     println!("I immutably borrowed {} - {} edition", book.title, book.year);
}

// Cette fonction prend une référence mutable d'une instance de la
// structure `Book` en paramètre, et initialise sa date de publication
// à 2014.
fn new_edition(book: &mut Book) {
     book.year = 2014;
     println!("I mutably borrowed {} - {} edition", book.title, book.year);
}

fn main() {
    // On créé une instance immuable de la
    // structure `Book` nommée `immutabook`.
    let immutabook = Book {
        // Les chaînes littérales sont typées `&'static str`.
        author: "Douglas Hofstadter",
        title: "Gödel, Escher, Bach",
        year: 1979,
    };

    // On créé une copie mutable de `immutabook` nommée
    // `mutabook`.
    let mut mutabook = immutabook;

    // Emprunte un objet en lecture seule.
    borrow_book(&immutabook);

    // Emprunte un objet en lecture seule.
    borrow_book(&mutabook);

    // Récupère une référence mutable d'un objet mutable.
    new_edition(&mut mutabook);

    // Erreur! Vous ne pouvez pas récupérer une référence
    // mutable d'un objet immuable.
    // new_edition(&mut immutabook);
    // FIXME ^ Décommentez cette ligne.
}
```

Voir aussi

La lifetime 'static.

13-3-2 - Verrouillage des ressources

Lorsqu'une référence immuable d'une ressource est récupérée, cette dernière est également « gelée », « verrouillée ». Lorsqu'une ressource est gelée, l'objet initial (celui à partir duquel une référence a été récupérée) ne peut être modifié jusqu'à ce que toutes les références soient détruites(i.e. ne figurent plus dans le contexte).

```rust
fn main() {
    let mut _mutable_integer = 7i32;

    {
        // On emprunte `_mutable_integer`.
        // Accès en lecture uniquement.
        let _large_integer = &_mutable_integer;

        // Erreur! `_mutable_integer` est gelé dans ce contexte
         // (i.e. la ressource est empruntée).
         // _mutable_integer = 50;
         // FIXME ^ Décommentez/commentez cette ligne.

         // On sort du contexte de `_large_integer`.
     }

     // Aucun problème! `_mutable_integer` n'est plus gelé dans ce contexte.
     _mutable_integer = 3;
}
```

13-3-3 - Limitations des accès lecture/écriture

Il est possible de faire autant « d'emprunts immuables » (i.e. récupérer une référence immuable) qu'on le souhaite. Toutefois, tant qu'il y a des accès en lecture par référence, l'objet original ne peut pas être modifié. Par contre, un seul accès en écriture, par ressource, est permis tant que le dernier emprunt par référence mutable n'est pas terminé (i.e. tant que la référence mutable est toujours dans le contexte).

```rust
struct Point { x: i32, y: i32, z: i32 }

fn main() {
    let mut point = Point { x: 0, y: 0, z: 0 };

    {
        let borrowed_point = &point;
        let another_borrow = &point;

         // Vous pouvez accéder à la ressource par le biais des références
         // créées et par l'objet original.
         println!("Point has coordinates: ({}, {}, {})",
                  borrowed_point.x, another_borrow.y, point.z);

         // Erreur! Vous ne pouvez pas avoir des accès en écriture à une ressource
         // qui est déjà empruntée par une référence immuable (i.e. accès en lecture).
         // let mutable_borrow = &mut point;
         // TODO ^ Essayez de décommenter cette ligne.

         // On sort du contexte des références
         // immuables.
     }

     {
         let mutable_borrow = &mut point;

         // On modifie la ressource par le biais d'une
         // référence mutable.
         mutable_borrow.x = 5;
         mutable_borrow.y = 2;
         mutable_borrow.z = 1;
```

```

        // Erreur! Vous ne pouvez pas accéder à `point` en lecture
        // alors que la ressource est potentiellement en train d'être modifiée.
        // let y = &point.y;
        // TODO ^ Essayez de décommenter cette ligne.

        // Erreur! On ne peut pas afficher `point.z` en utilisant
        // la macro `println!` car elle prend en paramètre une référence
        // immuable.
        // println!("Point Z coordinate is {}", point.z);
        // TODO ^ Essayez de décommenter cette ligne.

        // Ok! Les références mutables peuvent être passées comme
        // références immuables à `println!`.
        println!("Point has coordinates: ({}, {}, {})",
                 mutable_borrow.x, mutable_borrow.y, mutable_borrow.z);

        // On sort du contexte des références
        // mutables.
    }

    // Les accès en lecture sur `point` sont de nouveau
    // permis par le vérificateur d'emprunts.
    let borrowed_point = &point;
    println!("Point now has coordinates: ({}, {}, {})",
             borrowed_point.x, borrowed_point.y, borrowed_point.z);
}
```

13-3-4 - Le pattern ref

Lorsque vous vous servez du pattern matching ou de la déstructuration dans une assignation (let), le mot-clé ref peut être utilisé pour récupérer une référence d'un (ou plusieurs) champ d'une structure et/ou d'un tuple. Le code ci-dessous propose des exemples où ce modèle peut être utile :

```
#[derive(Clone, Copy)]
struct Point { x: i32, y: i32 }

fn main() {
    let c = 'Q';

    // Un emprunt effectué avec le mot-clé `ref` (placé dans la l-value)
    // est équivalent à une assignation "C-like" avec le `&` (placé dans la r-value).
    let ref ref_c1 = c;
     let ref_c2 = &c;

     println!("ref_c1 equals ref_c2: {}", *ref_c1 == *ref_c2);

     let point = Point { x: 0, y: 0 };

     // `ref` peut également être utilisé lors de la déstructuration
     // d'une structure (aussi valable pour les tuples(structures et littéraux)).
     let _copy_of_x = {
         // `ref_to_x` est une référence du champ `x` contenu dans
         // l'instance `point`.
         let Point { x: ref ref_to_x, y: _ } = point;

         // Renvoie une copie du champ `x` de l'instance `point`.
         *ref_to_x
     };

     // Copie mutable de l'instance `point`.
     let mut mutable_point = point;

     {
         // `ref` peut être combiné avec `mut` pour récupérer
         // une référence mutable.
         let Point { x: _, y: ref mut mut_ref_to_y } = mutable_point;

         // On modifie le champ `y` de l'instance `mutable_point` par
```

```
        // le biais de la référence mutable qu'on a récupéré.
        *mut_ref_to_y = 1;
    }

    println!("point is ({}, {})", point.x, point.y);
    println!("mutable_point is ({}, {})", mutable_point.x, mutable_point.y);

    // Un tuple mutable qui contient un pointeur et un entier non-signé
    // codé sur 32 bits.
    let mut mutable_tuple = (Box::new(5u32), 3u32);

    {
        // Nous déstructurons le tuple `mutable_tuple` pour modifier
        // la valeur de `last` (dernière élément indexé par le tuple).
        let (_, ref mut last) = mutable_tuple;
        *last = 2u32;
        // `_` est ajouté dans la valeur de gauche car on ne souhaite
        // pas en récupérer le contenu.
    }

    println!("tuple is {:?}", mutable_tuple);
}
```

13-4 - Système de durée de vie

Une « lifetime » est une construction utilisée par le compilateur pour s'assurer que tous les emprunts (i.e. références immuables/mutables) sont valides. Plus précisément, la durée de vie d'une variable commence lorsqu'elle est créée et se termine une fois cette dernière détruite.

Note : Bien que les lifetimes et les contextes soient très souvent cités ensembles cela reste, malgré tout, deux concepts bien distincts.

```
// La création et destruction des lifetimes sont illustrées ci-dessous par des lignes.
// `i` possède la plus grande lifetime car son contexte englobe
// `borrow1` et `borrow2`. La durée de vie de `borrow1` ne peut pas être
// comparée à celle de `borrow2` car elle ne se trouve pas dans le
// même contexte.
fn main() {
    let i = 3; // Lifetime for `i` starts. ────────────────┐
    //                                                      │
    { //                                                    │
        let borrow1 = &i; // `borrow1` lifetime starts. ──┐│
        //                                                ││
        println!("borrow1: {}", borrow1); //              ││
    } // `borrow1 ends. ──────────────────────────────────┘│
    //                                                      │
    //                                                      │
    { //                                                    │
        let borrow2 = &i; // `borrow2` lifetime starts. ──┐│
        //                                                ││
        println!("borrow2: {}", borrow2); //              ││
    } // `borrow2` ends. ─────────────────────────────────┘│
    //                                                      │
}   // Lifetime ends. ─────────────────────────────────────┘
```

Notez qu'aucun nom ou type n'est assigné aux labels de lifetimes. Nous allons apprendre, dans la prochaine section, à nous servir des durées de vie (et appréhender l'utilisation des labels).

13-4-1 - Les labels

Le compilateur se sert des « annotations explicites » (labels) pour déterminer la durée de validité d'une référence. Dans le cas où les lifetimes ne peuvent pas être omises (**note** : Elles peuvent l'être grâce au concept **d'élision**), Rust dispose d'annotations explicites (labels) pour déterminer quelle devrait être la durée de vie d'une référence. Les labels sont précédés d'une `'apostrophe` :

```
foo<'a>
// `foo` dispose de la durée de vie `'a`.
```

Tout comme **les closures**, pour utiliser les labels, vous devrez avoir recours à la généricité. De plus, cette syntaxe permet de préciser que la durée de vie de `foo` ne peut pas excéder celle de `'a`.

Voici la syntaxe des labels lorsqu'ils sont appliqués à un type : `&'a T` (ou `&'a mut T`) où `'a` est un label déclaré près de l'identificateur de la fonction.

Si vous devez déclarer plusieurs lifetimes, la syntaxe est tout aussi simple :

```
foo<'a, 'b>
// `foo` dispose des lifetimes `'a` et `'b`.
```

Dans ce cas, la durée de vie de `foo` ne peut pas excéder la lifetime `'a` ou `'b`.

```
// `print_refs` prend deux références d'entiers possédant une durée de
// vie différente chacune (i.e. `'a` et `'b`). Ces deux lifetimes doivent être
// au moins aussi longues que celle de la fonction `print_refs`.
fn print_refs<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("x is {} and y is {}", x, y);
}

// Une fonction qui ne prend aucun argument, mais qui déclare quand même
// une lifetime `'a`.
fn failed_borrow<'a>() {
    let _x = 12;

    // ERREUR: `_x` ne vit pas assez longtemps.
    // let y: &'a i32 = &_x;
    // Tenter d'utiliser la lifetime `'a` sur une ressource au sein de la fonction
    // ne fonctionnera pas car la durée de vie de `&_x` est plus courte que celle
    // de `y`.
    // Une courte durée de vie ne peut être rallongée en cours de route.
}

fn main() {
    // On crée ces variables pour les emprunter
    // un peu plus bas.
    let (four, nine) = (4, 9);

    // `print_refs` emprunte (`&`) les deux ressources précédemment
    // créées.
    print_refs(&four, &nine);
    // Une ressource empruntée doit survivre à la fonction qui l'emprunte.
    // Autrement dit, la durée de vie de `four` et `nine` doit être
    // plus longue que celle de `print_refs`.

    failed_borrow();

    // `failed_borrow` ne contient aucune référence qui force la lifetime `'a` à
    // être plus longue que celle de la fonction, mais `'a` reste plus longue.
    // Parce que la durée de vie d'une ressource (empruntée) n'est jamais imposée,
    // la durée de vie par défaut est `'static`.
}
```

Voir aussi

La généricité et **les closures**.

13-4-2 - Les fonctions

Lorsque le concept **d'élision** ne peut pas être appliqué, les signatures de fonctions comportant des (labels de) lifetimes sont régies par quelques règles :

1. Toute référence doit être annotée d'une lifetime ;
2. Toute référence renvoyée doit posséder la même lifetime qu'en entrée ou la lifetime 'static.

Notez également que, si une fonction n'a pas de références en entrée, le renvoi de références est interdit car cela pourrait conduire à l'utilisation de ressources invalides.

```
// La référence passée en paramètre possédant la lifetime `'a`
// doit vivre au moins aussi longtemps que le fonction.
fn print_one<'a>(x: &'a i32) {
    println!("`print_one`: x is {}", x);
}

// L'utilisation des références mutables est également possible
// avec les lifetimes.
fn add_one<'a>(x: &'a mut i32) {
    *x += 1;
}

// Plusieurs entrées avec différentes lifetimes. Dans ce cas,
// il serait plus simple, pour les deux paramètres, d'avoir
// la même durée de vie (`'a`), mais dans des cas plus délicats,
// plusieurs lifetimes peuvent être requises.
fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x is {}, y is {}", x, y);
}

// Renvoyer des références qui ont été passées en paramètre est légal.
// Toutefois, veillez à renvoyer la bonne lifetime.
fn pass_x<'a, 'b>(x: &'a i32, _: &'b i32) -> &'a i32 { x }

// fn invalid_output<'a>() -> &'a i32 { &7 }
// La déclaration ci-dessus est invalide: `'a` doit, au moins,
// survivre à la fonction. Ici, `&7` créerait un entier et récupérerait
// sa référence. La ressource serait alors perdue une fois l'exécution
// de la fonction terminée, renvoyant une référence d'une ressource qui n'existe plus.

fn main() {
    let x = 7;
    let y = 9;

    print_one(&x);
    print_multi(&x, &y);

    let z = pass_x(&x, &y);
    print_one(z);

    let mut t = 3;
    add_one(&mut t);
    print_one(&t);
}
```

Voir aussi

Les fonctions.

13-4-3 - Les méthodes

Les méthodes sont annotées de la même manière que les fonctions :

```
struct Owner(i32);

impl Owner {
    // On déclare et annote les lifetimes comme
    // dans une fonction traditionnelle.
    fn add_one<'a>(&'a mut self) { self.0 += 1; }
    fn print<'a>(&'a self) {
```

```
        println!("`print`: {}", self.0);
    }
}

fn main() {
    let mut owner  = Owner(18);

    owner.add_one();
    owner.print();
}
```

Voir aussi

Les méthodes.

13-4-4 - Les structures

La déclaration des labels pour les structures ne diffère pas beaucoup non plus de celle des fonctions :

```
// On créé un type nommé `Borrowed` qui a pour attribut
// une référence d'un entier codé sur 32 bits. La référence
// doit survivre à l'instance de la structure `Borrowed`.
#[derive(Debug)]
struct Borrowed<'a>(&'a i32);

// Même combat, ces deux références doivent survivre à l'instance
// (ou aux instances)  de la structure `NamedBorrowed`.
#[derive(Debug)]
struct NamedBorrowed<'a> {
    x: &'a i32,
    y: &'a i32,
}

// On créé une énumération qui contient deux variantes :
// 1. Un tuple qui prend en entrée un entier codé sur 32 bits;
// 2. Un tuple qui prend en entrée une référence d'un `i32`.
#[derive(Debug)]
enum Either<'a> {
    Num(i32),
    Ref(&'a i32),
}

fn main() {
    let x = 18;
    let y = 15;

    let single = Borrowed(&x);
    let double = NamedBorrowed { x: &x, y: &y };
    let reference = Either::Ref(&x);
    let number    = Either::Num(y);

    println!("x is borrowed in {:?}", single);
    println!("x and y are borrowed in {:?}", double);
    println!("x is borrowed in {:?}", reference);
    println!("y is *not* borrowed in {:?}", number);
}
```

Voir aussi

Les structures.

13-4-5 - Les restrictions

Tout comme les types génériques, les lifetimes (elles-mêmes génériques) utilisent les restrictions. Ici, le caractère : a une signification quelque peu différente, mais + possède la même fonction. La déclaration se lit comme suit :

1. T: 'a : Toutes les références dans le type T doivent, au moins, survivre à la lifetime 'a ;
2. T: Trait + 'a : Le type T doit implémenter le trait Trait et toutes les références doivent survivre à la lifetime 'a.

L'exemple ci-dessous illustre les explications précédentes :

```
use std::fmt::Debug; // On importe le trait `Debug`.
// Trait avec lequel nous allons filtrer les entrées
// des fonctions génériques.

#[derive(Debug)]
struct Ref<'a, T: 'a>(&'a T);
// `Ref` contient la référence d'un type générique `T` qui possède
// une lifetime inconnue nommée `'a`. Toutes les références contenues par
// le type `T` sont contraintes à survivre à la lifetime `'a`. De plus,
// la durée de vie de `Ref` ne peut pas excéder la lifetime `'a`.

// Une fonction générique qui affiche le résultat de l'utilisation
// du trait `Debug`.
fn print<T>(t: T) where
    T: Debug {
    println!("`print`: t is {:?}", t);
}

// Ici, nous avons une référence de type `T` en entrée où
// `T` implémente le trait `Debug` et toutes les références
// contenues par le type `T` survivent à la lifetime `'a`. La lifetime
// `'a` doit survivre à l'appel de la fonction.
fn print_ref<'a, T>(t: &'a T) where
    T: Debug + 'a {
    println!("`print_ref`: t is {:?}", t);
}

fn main() {
    let x = 7;
    let ref_x = Ref(&x);

    print_ref(&ref_x);
    print(ref_x);
}
```

Voir aussi

La généricité, **les restrictions** et **les restrictions multiples**.

13-4-6 - La coercition

Une longue durée de vie peut être raccourcie afin d'opérer dans un contexte où elle ne fonctionnerait pas en temps normal. Cette opération peut être effectuée par inférence du compilateur ou en déclarant une différence de durée de vie(e.g. l'une est, au moins, aussi longue que l'autre, sinon plus).

```
// Ici, Rust va inférer une durée de vie aussi courte que possible.
// Les deux références sont alors assignées à cette durée de vie.
fn multiply<'a>(first: &'a i32, second: &'a i32) -> i32 {
    first * second
}

// `<'a: 'b, 'b>` se lit comme suit: La lifetime `'a` est au moins aussi longue
// que `'b`. Ici, nous prenons en entrée un entier soumis à la lifetime
// `&'a i32` et renvoyons un entier (le même entier, en fait) soumis à la lifetime
```

```
// `&'b i32` comme résultat de la coercition.
// Note: Nous pouvons renvoyer l'entier taggé avec la lifetime `'b` car `'a` et `'b`
// ont une durée de vie aussi longue, pour le temps de l'exécution de la fonction,
// tout du moins.
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}

fn main() {
    let first = 2; // Longue lifetime.

    {
        let second = 3; // Courte lifetime.

        // Note: Ici `first` et `second` possèdent la même durée de vie.
        println!("The product is {}", multiply(&first, &second));
        println!("{} is the first", choose_first(&first, &second));
    };
}
```

13-4-7 - La lifetime 'static

Une ressource annotée du label 'static possède la plus longue durée de vie qu'on puisse assigner. La durée de vie de 'static est égale à celle du programme lui-même et peut également être raccourcie selon le contexte.

Il y a deux façons d'assigner la lifetime 'static, et toutes deux stockeront la ressource directement dans le binaire en lecture seule :

1. Créer une constante avec la déclaration static ;
2. Créer une chaîne de caractères avec le "littéral" en l'annotant du type &'static str.

Voici un exemple pour illustrer les deux manières de faire :

```
// On créé une constante avec la lifetime `'static` en utilisant
// le mot-clé `static`.
static NUM: i32 = 18;

// Renvoie une référence de `NUM` où sa lifetime `'static`
// est obligée de s'aligner avec la durée de vie du paramètre
// passé à la fonction.
fn coerce_static<'a>(_: &'a i32) -> &'a i32 {
    &NUM
}

fn main() {
    {
        // On créé une chaîne de caractères littérale, primitive
        // et on l'affiche.
        let static_string = "I'm in read-only memory";
        println!("static_string: {}", static_string);

        // Lorsque `static_string` sortira du contexte, la référence
        // ne pourra plus être utilisée, mais la ressource restera
        // présente dans le binaire.
    }

    {
        // On créé un entier à passer à la
        // fonction `coerce_static`:
        let lifetime_num = 9;

        // On aligne, adapte la durée de vie de `NUM à
        // celle de `lifetime_num`:
        let coerced_static = coerce_static(&lifetime_num);

        println!("coerced_static: {}", coerced_static);
    }
```

```
    println!("NUM: {} stays accessible!", NUM);
}
```

Voir aussi

Les constantes.

13-4-8 - Annotation implicite

En règle générale, décrire explicitement la durée de vie d'une référence n'est pas nécessaire et nous préférerons passer la main au compilateur, qui se chargera de nous épargner l'écriture des annotations et d'améliorer la lisibilité du code. Ce processus d'annotation implicite se nomme **l'élision**. Il s'agit ici d'omettre volontairement les annotations pour que le compilateur le fasse à notre place. L'élision ne peut être appliquée que lorsqu'un pattern de durée de vie est commun, simple à deviner.

Le code source qui suit présente quelques exemples où nous avons volontairement omis les annotations. Pour une description plus exhaustive du concept d'élision, n'hésitez pas à consulter la section **lifetime elision** du livre.

```
// `elided_input` et `annotated_input` ont fondamentalement la même signature,
// sauf que la lifetime de l'entrée de la fonction `elided_input` a été omise
// et ajoutée par le compilateur.
fn elided_input(x: &i32) {
    println!("`elided_input`: {}", x)
}

fn annotated_input<'a>(x: &'a i32) {
    println!("`annotated_input`: {}", x)
}

// Même combat, `elided_pass` et `annotated_pass` possèdent la même signature
// sauf que la lifetime de `x`, pour la fonction `elided_pass`, a été omise
// et ajoutée par le compilateur. Annotation implicite.
fn elided_pass(x: &i32) -> &i32 { x }

fn annotated_pass<'a>(x: &'a i32) -> &'a i32 { x }

fn main() {
    let x = 3;

    elided_input(&x);
    annotated_input(&x);

    println!("`elided_pass`: {}", elided_pass(&x));
    println!("`annotated_pass`: {}", annotated_pass(&x));
}
```

Voir aussi

Le chapitre du livre sur l'élision.

14 - Les traits

Un trait est un agrégat de méthodes définies pour un type inconnu : Self. Elles peuvent accéder aux autres méthodes déclarées dans le même trait.

Les traits peuvent être implémentés pour n'importe quel type de donnée. Dans l'exemple ci-dessous, nous définissons Animal, un groupe de méthodes. Le trait Animal est alors implémenté pour le type Sheep, permettant l'utilisation des méthodes de Animal avec une instance du type Sheep.

```
struct Sheep { naked: bool, name: &'static str }
```

```

trait Animal {
    // Méthode statique; `Self` fait référence au type ayant implémenté
    // le trait.
    fn new(name: &'static str) -> Self;

    // Méthode d'instance; Elles renverront une chaîne de caractères.
    fn name(&self) -> &'static str;
     fn noise(&self) -> &'static str;

     // Les traits peuvent fournir une implémentation par défaut.
     fn talk(&self) {
         println!("{} says {}", self.name(), self.noise());
     }
 }

 impl Sheep {
     fn is_naked(&self) -> bool {
         self.naked
     }

     fn shear(&mut self) {
         if self.is_naked() {
             // Les méthodes de `Self` peuvent utiliser les méthodes déclarées
             // par le trait.
             println!("{} is already naked...", self.name());
         } else {
             println!("{} gets a haircut!", self.name);

             self.naked = true;
         }
     }
 }

 // Implémentation des services du trait `Animal`
 // pour le type `Sheep`.
 impl Animal for Sheep {
     // En l'occurrence, `Self` fait référence à `Sheep`.
     fn new(name: &'static str) -> Sheep {
         Sheep { name: name, naked: false }
     }

     fn name(&self) -> &'static str {
         self.name
     }

     fn noise(&self) -> &'static str {
         if self.is_naked() {
             "baaaaah?"
         } else {
             "baaaaah!"
         }
     }

     // L'implémentation par défaut fournie par le trait
     // peut être réécrite.
     fn talk(&self) {
         // Par exemple, nous pourrions fournir une description plus précise.
         println!("{} pauses briefly... {}", self.name, self.noise());
     }
 }

 fn main() {
     // Typer l'identificateur est nécessaire dans ce cas de figure.
     let mut dolly: Sheep = Animal::new("Dolly");
     // TODO ^ Essayez de supprimer le type annoté.

     dolly.talk();
     dolly.shear();
     dolly.talk();
 }
```

14-1 - L'attribut Derive

Le compilateur est capable de fournir de simples implémentations pour cetains traits par le biais de **l'attribut** #[derive]. Ces traits peuvent toujours être implémentés manuellement si un traitement plus complexe est attendu.

Voici une liste de traits pouvant être dérivés :

- Les traits de comparaison : **Eq**, **PartialEq**, **Ord**, **PartialOrd**;
- **Clone**, pour créer une instance (T) à partir d'une référence (&T) par copie ;
- **Copy**, pour permettre à un type d'être copié plutôt que transféré ;
- **Hash**, pour générer un hash depuis &T ;
- **Debug**, pour formater une valeur en utilisant le formatteur {:?}.

```
// `Centimeters` est un tuple qui peut être comparé.
#[derive(PartialEq, PartialOrd)]
struct Centimeters(f64);

// `Inches` est un tuple qui peut être affiché.
#[derive(Debug)]
struct Inches(i32);

impl Inches {
    fn to_centimeters(&self) -> Centimeters {
        let &Inches(inches) = self;

        Centimeters(inches as f64 * 2.54)
    }
}

// `Seconds` est un tuple ne possédant aucun attribut.
struct Seconds(i32);

fn main() {
    let _one_second = Seconds(1);

    // Erreur: `Seconds` ne peut pas être affiché; Il n'implémente pas le trait `Debug`.
    // println!("One second looks like: {:?}", _one_second);
    // TODO ^ Essayez de décommenter cette ligne.

    // Erreur: `Seconds` ne peut pas être comparé; Il n'implémente pas le trait `PartialEq`.
    // let _this_is_true = (_one_second == _one_second);
    // TODO ^ Essayez de décommenter cette ligne.

    let foot = Inches(12);

    println!("One foot equals {:?}", foot);

    let meter = Centimeters(100.0);

    let cmp =
        if foot.to_centimeters() < meter {
            "smaller"
        } else {
            "bigger"
        };

    println!("One foot is {} than one meter.", cmp);
}
```

Voir aussi

Derive.

14-2 - La surcharge des opérateurs

Avec Rust, nombre d'opérateurs peuvent être surchargés via les traits. En d'autres termes, des opérateurs peuvent être utilisés pour accomplir différentes tâches en fonction des arguments passés en entrée. Cette manipulation est possible parce que les opérateurs sont des sucres syntaxes visant à masquer l'appel des méthodes liées à ces derniers. Par exemple, l'opérateur `+` dans l'expression `a + b` appelle la méthode `add` (`a.add(b)`). La méthode `add` appartient au trait `Add`, d'où l'utilisation de l'opérateur `+` par tous les types implémentant le trait.

Vous pouvez retrouver la liste des traits surchargeant des opérateurs [ici][operators].

```rust
use std::ops;

struct Foo;
struct Bar;

#[derive(Debug)]
struct FooBar;

#[derive(Debug)]
struct BarFoo;

// Le trait `std::ops::Add` est utilisé pour permettre la surcharge de `+`.
// Ici, nous spécifions `Add<Bar>` - cette implémentation sera appelée si l'opérande de droite est
// de type `Bar`.
// Le bloc ci-dessous implémente l'opération : `Foo + Bar = FooBar`.
impl ops::Add<Bar> for Foo {
    type Output = FooBar;

    fn add(self, _rhs: Bar) -> FooBar {
        println!("> Foo.add(Bar) was called");

        FooBar
    }
}

// En inversant les types, nous nous retrouvons à implémenter une addition non-commutative.
// Ici, nous spécifions `Add<Foo>` - cette implémentation sera appelée si l'opérande de droite
// est de type `Foo`.
// Le bloc ci-dessous implémente l'opération: `Bar + Foo = BarFoo`.
impl ops::Add<Foo> for Bar {
    type Output = BarFoo;

    fn add(self, _rhs: Foo) -> BarFoo {
        println!("> Bar.add(Foo) was called");

        BarFoo
    }
}

fn main() {
    println!("Foo + Bar = {:?}", Foo + Bar);
    println!("Bar + Foo = {:?}", Bar + Foo);
}
```

Voir aussi

Add, **index de la syntaxe**.

14-3 - Le trait Drop

Le trait **Drop** ne possède qu'une seule méthode : `drop` ; cette dernière est automatiquement appelée lorsqu'un objet sort du contexte. La fonction principale du trait `Drop` est de libérer les ressources que les instances, du type ayant implémenté le trait, possèdent.

Box, Vec, String, File et Process sont autant d'exemples de types qui implémentent le trait Drop pour libérer leurs ressources. Le trait Drop peut également être implémenté manuellement pour répondre aux besoins de vos propres types.

Dans l'exemple suivant, nous affichons quelque chose dans la console à partir de la méthode drop pour notifier chaque appel.

```
struct Droppable {
    name: &'static str,
}

// Implémentation basique de `drop` qui affiche un message dans la console.
impl Drop for Droppable {
    fn drop(&mut self) {
        println!("> Dropping {}", self.name);
    }
}

fn main() {
    let _a = Droppable { name: "a" };

    // block A
    {
        let _b = Droppable { name: "b" };

        // block B
        {
            let _c = Droppable { name: "c" };
            let _d = Droppable { name: "d" };

            println!("Exiting block B");
        }
        println!("Just exited block B");

        println!("Exiting block A");
    }
    println!("Just exited block A");

    // La variable peut être libérée manuellement en utilisant la fonction `(std::mem::)drop`.
    drop(_a);
    // TODO ^ Essayez de commenter cette ligne.

    println!("end of the main function");

    // `_a` ne *sera pas* libérée une seconde fois ici puisque nous l'avons déjà fait
    // plus haut, manuellement.
}
```

14-4 - Les itérateurs

Le trait Iterator est utilisé pour implémenter les itérateurs sur les collections tels que les tableaux.

Le trait nécessite seulement la définition d'une méthode pour l'élément next. Elle peut être implémentée manuellement en utilisant un bloc impl ou automatiquement (comme pour les tableaux et intervalles).

Pour les utilisations les plus communes, la boucle for peut convertir certaines collections en itérateurs en utilisant la méthode **.into_iterator()**.

Les méthodes pouvant être appelées en utilisant le trait Iterator, en plus de celles présentées dans l'exemple ci-dessous, sont listées **ici**.

```
struct Fibonacci {
    curr: u32,
    next: u32,
```

```
}

// Implémentation de `Iterator` pour le type `Fibonacci`.
// Le trait `Iterator` nécessite l'implémentation d'une méthode seulement pour l'élément `next`.
impl Iterator for Fibonacci {
    type Item = u32;

    // Ici, nous définissons la séquence utilisant `.curr` et `.next`.
    // Le type de renvoi est `Option<T>`:
    //     * Lorsque l'`Iterator` est terminé, `None` est renvoyé;

    //     * Autrement, la valeur suivante est enveloppé dans une instance `Some` et renvoyée.
    fn next(&mut self) -> Option<u32> {
        let new_next = self.curr + self.next;

        self.curr = self.next;
        self.next = new_next;

        // Puisqu'il n'y a pas de limite à une suite de Fibonacci, l'`Iterator`
        // ne renverra jamais `None`.
        Some(self.curr)
    }
}

// Renvoie un générateur de suites de Fibonacci.
fn fibonacci() -> Fibonacci {
    Fibonacci { curr: 1, next: 1 }
}

fn main() {
    // `0..3` est un `Iterator` qui génère: 0, 1, et 2 (i.e. intervalle `[0, 3[`).
    let mut sequence = 0..3;

    println!("Four consecutive `next` calls on 0..3");
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());
    println!("> {:?}", sequence.next());


    // `for` parcourt un `Iterator` jusqu'à ce qu'il renvoie `None`.
    // Chaque valeur contenue par un objet `Some` est libérée de son conteneur
    // puis assignée à une variable (en l'occurrence, `i`).
    println!("Iterate through 0..3 using `for`");
    for i in 0..3 {
        println!("> {}", i);
    }

    // La méthode `take(n)` réduit un `Iterator` à ces `n` premiers éléments.
    println!("The first four terms of the Fibonacci sequence are: ");
    for i in fibonacci().take(4) {
        println!("> {}", i);
    }

    // La méthode `skip(n)` tronque les `n` premiers éléments d'un `Iterator`.
    println!("The next four terms of the Fibonacci sequence are: ");
    for i in fibonacci().skip(4).take(4) {
        println!("> {}", i);
    }

    let array = [1u32, 3, 3, 7];

    // La méthode `iter` construit un `Iterator` sur un(e) tableau/slice.
    println!("Iterate the following array {:?}", &array);
    for i in array.iter() {
        println!("> {}", i);
    }
}
```

14-5 - Le trait Clone

Lors du traitement des ressources, le comportement par défaut est de les transférer lors d'un assignement ou un appel de méthode. Cependant, il est parfois nécessaire d'effectuer une copie des ressources en question.

C'est exactement la fonction du trait **Clone**, qui nous permettra d'utiliser la méthode clone().

```
// Une structure unitaire sans ressources.
#[derive(Debug, Clone, Copy)]
struct Nil;

// Un tuple avec des ressources qui implémente le trait `Clone`.
#[derive(Clone, Debug)]
struct Pair(Box<i32>, Box<i32>);

fn main() {
    // Un instancie `Nil`.
    let nil = Nil;
    // On copie l'objet `Nil`, aucune ressource à transférer.
    let copied_nil = nil;

    // Les deux instances peuvent être utilisées indépendament l'une de l'autre.
    println!("original: {:?}", nil);
    println!("copy: {:?}", copied_nil);

    // On crée un objet `Pair`.
    let pair = Pair(Box::new(1), Box::new(2));
    println!("original: {:?}", pair);

    // On copie `pair` dans `moved_pair`, transfert de ressources.
    let moved_pair = pair;
    println!("copy: {:?}", moved_pair);

    // Erreur! `pair` ne possède plus ses ressources.
    // println!("original: {:?}", pair);
    // TODO ^ Essayez de décommenter cette ligne.

    // On copie `moved_pair` dans `cloned_pair` (ressources incluses).
    let cloned_pair = moved_pair.clone();
    // On libère la ressource originale avec `std::mem::drop`.
    drop(moved_pair);

    // Erreur! `moved_pair` a été libérée.
    // println!("copy: {:?}", moved_pair);
    // TODO ^ Essayez de décommenter cette ligne.

    // La copie obtenue par `.clone()` peut toujours être utilisée !
    println!("clone: {:?}", cloned_pair);
}
```

15 - macro_rules!

Rust fournit un puissant système de macros qui permet la **métaprogrammation**. Comme vous l'avez vu dans les chapitres précédents, les macros ressemblent aux fonctions, excepté que leurs identificateurs se terminent par un point d'exclamation !, mais au lieu de générer un appel de fonction, les macros sont étendues dans le code source et compilées avec le reste du programme.

Il est possible de créer une macro en utilisant la macro macro_rules!.

```
// Une simple macro nommée `say_hello`.
macro_rules! say_hello {
    // `()` indique que la macro ne prend aucun argument.
    () => (
        // La macro étendra le contenu de ce bloc.
        println!("Hello!");
```

```
    )
}

fn main() {
    // Cet appel va étendre `println!("Hello");`.
    say_hello!()
}
```

15-1 - Les indicateurs

Les arguments d'une macro sont préfixés par un `$` et sont typés par le biais d'un indicateur :

```
macro_rules! create_function {
    // Cette macro prend un argument possédant l'indicateur `ident`
    // et créé une fonction nommée `$func_name`.
    // L'indicateur `ident` est utilisé pour les identificateur de variables/fonctions.
    ($func_name:ident) => (
        fn $func_name() {
            // La macro `stringify!` convertit un `ident` en chaîne de caractères.
            println!("You called {:?}()",
                     stringify!($func_name))
        }
    )
}

// On créé les fonctions `foo` et `bar` avec la macro ci-dessus.
create_function!(foo);
create_function!(bar);

macro_rules! print_result {
    // Cette macro prend une expression de type `expr` et
    // affiche sa représentation sous forme de chaîne de caractères
    // ainsi que son résultat.
    ($expression:expr) => (
        // `stringify!` va convertir l'expression *telle qu'elle est* dans une chaîne
        // de caractères.
        println!("{:?} = {:?}",
                 stringify!($expression),
                 $expression)
    )
}

fn main() {
    foo();
    bar();

    print_result!(1u32 + 1);

    // Rappelez-vous que les blocs sont également des expressions !
    print_result!({
        let x = 1u32;

        x * x + 2 * x - 1
    });
}
```

Voici une liste exhaustive des indicateurs existants :

- `block` ;
- `expr`, pour les expressions ;
- `ident`, pour les identificateurs de variables et fonctions ;
- `item` ;
- `pat` (pattern) ;
- `path` ;
- `stmt` (déclaration) ;
- `token` (token tree) ;

- ty (type).

15-2 - Surcharge

Les macros peuvent être surchargées pour accepter différentes combinaisons d'arguments. Dans cet esprit, macro_rules! peut fonctionner de la même manière qu'un bloc match :

```
// `test!` va comparer `$left` et `$right` de différentes
// manières suivant son utilisation à l'invocation:
macro_rules! test {
    // Il n'est pas nécessaire de séparer les arguments par des virgules.
    // N'importe quel modèle peut être utilisé !
    ($left:expr; and $right:expr) => (
        println!("{:?} and {:?} is {:?}",
                 stringify!($left),
                 stringify!($right),
                  $left && $right)
     );
     // ^ Chaque branche doit se terminer par un point-virgule.
     ($left:expr; or $right:expr) => (
         println!("{:?} or {:?} is {:?}",
                  stringify!($left),
                  stringify!($right),
                  $left || $right)
     );
}

fn main() {
    test!(1i32 + 1 == 2i32; and 2i32 * 2 == 4i32);
    test!(true; or false);
}
```

15-3 - Répétition

Les macros peuvent utiliser le quantificateur + dans la liste des arguments pour indiquer qu'un argument peut être répété au moins une(1) fois ou * pour indiquer que l'argument peut être répété zéro(0) ou plusieurs fois.

Dans l'exemple suivant, entourer le matcher avec $(...),+ va permettre la capture d'une ou plusieurs expressions, séparées par des virgules. Notez également que le point-virgule est optionnel dans le dernier cas (i.e. la dernière expression capturée).

```
// `min!` va calculer le minimum entre chaque argument passé, peu importe le nombre.
macro_rules! find_min {
    // Expression de base:
    ($x:expr) => ($x);
    // `$x` suivi par au moins un `$y,`.
    ($x:expr, $($y:expr),+) => (
        // On appelle `find_min` sur les $y suivants`.
        std::cmp::min($x, find_min!($($y),+))
    )
}

fn main() {
    println!("{}", find_min!(1u32));
    println!("{}", find_min!(1u32 + 2 , 2u32));
    println!("{}", find_min!(5u32, 2u32 * 3, 4u32));
}
```

15-4 - DRY (Don't Repeat Yourself)

Les macros permettent l'écriture de code DRY en recyclant les parties communes d'un ensemble de fonctions/tests. Voici un exemple qui implémente et test les opérateurs +=, *= et -= sur la structure Vec<T> :

```
// Dans le fichier dry.rs
use std::ops::{Add, Mul, Sub};

macro_rules! assert_equal_len {
    // L'indicateur `tt` (token tree) est utilisé pour les opérateurs
    // et les tokens.
    ($a:ident, $b: ident, $func:ident, $op:tt) => (
        assert!($a.len() == $b.len(),
                "{:?}: dimension mismatch: {:?} {:?} {:?}",
                 stringify!($func),
                 ($a.len(),),
                 stringify!($op),
                 ($b.len(),));
     )
}

macro_rules! op {
     ($func:ident, $bound:ident, $op:tt, $method:ident) => (
         fn $func<T: $bound<T, Output=T> + Copy>(xs: &mut Vec<T>, ys: &Vec<T>) {
             assert_equal_len!(xs, ys, $func, $op);

             for (x, y) in xs.iter_mut().zip(ys.iter()) {
                 *x = $bound::$method(*x, *y);
                 // *x = x.$method(*y);
             }
         }
     )
}

// On implémente les fonctions `add_assign`, `mul_assign` et `sub_assign`.
op!(add_assign, Add, +=, add);
op!(mul_assign, Mul, *=, mul);
op!(sub_assign, Sub, -=, sub);

mod test {
    use std::iter;
    macro_rules! test {
        ($func: ident, $x:expr, $y:expr, $z:expr) => {
            #[test]
            fn $func() {
                for size in 0usize..10 {
                    let mut x: Vec<_> = iter::repeat($x).take(size).collect();
                    let y: Vec<_> = iter::repeat($y).take(size).collect();
                    let z: Vec<_> = iter::repeat($z).take(size).collect();

                    super::$func(&mut x, &y);

                    assert_eq!(x, z);
                }
            }
        }
    }

    // Test de `add_assign`, `mul_assign` et `sub_assign`.
    test!(add_assign, 1u32, 2u32, 3u32);
    test!(mul_assign, 2u32, 3u32, 6u32);
    test!(sub_assign, 3u32, 2u32, 1u32);
}
```

```
$ rustc --test dry.rs && ./dry
running 3 tests
test test::mul_assign ... ok
test test::add_assign ... ok
test test::sub_assign ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured
```

16 - La gestion des erreurs

La gestion d'erreur est le processus de gestion d'un échec potentiel. Par exemple, ne pas parvenir à lire un fichier et continuer à utiliser malgré tout cette mauvaise entrée serait clairement problématique. Cerner et gérer explicitement ces erreurs épargne le reste du programme d'un bon nombre de problèmes.

Pour une explicitation plus exhaustive à propos de la gestion des erreurs, référez-vous à la section dédiée à la gestion des erreurs **dans le livre officiel**.

16-1 - La macro panic

Le plus simple mécanisme de gestion d'erreur que nous allons voir est panic. Il affiche un message d'erreur, lance la tâche et généralement met fin au programme. Ici, nous appelons explicitement panic dans notre condition :

```
fn give_princess(gift: &str) {
    // Les princesses détestent les serpents, donc nous devons tout arrêter si elles n'acceptent pas !
    if gift == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", gift);
}

fn main() {
    give_princess("teddy bear");
    give_princess("snake");
}
```

16-2 - L'enum Option et la méthode unwrap

Dans le dernier exemple, nous avons montré qu'il était possible de mettre en échec le programme quand bon nous semble. Nous disions que notre programme pouvait "paniquer" si la princesse recevait un présent inapproprié - un serpent. Mais qu'en est-il du cas où la princesse attendait un cadeau mais n'en reçoit pas ? Ce cas de figure serait tout bonnement irrecevable, donc inutile d'être géré.

Nous pourrions tester ce cas contre une chaîne de caractères vide (""), comme nous l'avons fait avec le serpent. Puisque nous utilisons Rust, laissons plutôt le compilateur gérer les cas où il n'y pas de cadeau.

Une enum nommée Option<T> dans la bibliothèque standard est utilisée lorsque "l'absence de" est une possibilité. Elle-même est représentée par deux variantes :

- Some(T) : Un élément de type T a été trouvé ;
- None : Aucun élément n'a été trouvé.

Ces cas peuvent être explicitement gérés par le biais de match ou implicitement avec unwrap. La gestion implicite renverra l'élément contenu en cas de succès, sinon un panic sera lancé.

Notez que s'il est possible de personnaliser manuellement le message d'erreur de panic, ce n'est pas le cas pour unwrap qui nous laissera avec des informations moins intelligibles qu'une gestion explicite. Dans l'exemple suivant, la gestion explicite offre un plus grand contrôle sur le résultat tout en permettant l'utilisation de panic, si désiré.

```
// Le roturier a déjà tout vu et accepte de bon coeur n'importe quel présent.
// Tous les présents sont gérés explicitement en utilisant `match`.
fn give_commoner(gift: Option<&str>) {
    // On définit une action pour chaque cas.
    match gift {
        Some("snake") => println!("Yuck! I'm throwing that snake in a fire."),
        Some(inner)   => println!("{}? How nice.", inner),
        None          => println!("No gift? Oh well."),
```

```
    }
}

// Notre précieuse princesse va `panic` à la vue des serpents.
// Tous les présents sont gérés implicitement en utilisant `unwrap`.
fn give_princess(gift: Option<&str>) {
    // `unwrap` renvoie un `panic` lorsqu'il reçoit la variante `None`.
    let inside = gift.unwrap();
    if inside == "snake" { panic!("AAAaaaaa!!!!"); }

    println!("I love {}s!!!!!", inside);
}

fn main() {
    let food  = Some("cabbage");
    let snake = Some("snake");
    let void  = None;

    give_commoner(food);
    give_commoner(snake);
    give_commoner(void);

    let bird = Some("robin");
    let nothing = None;

    give_princess(bird);
    give_princess(nothing);
}
```

16-2-1 - Les combinateurs: map

match est une approche valable pour gérer les Options. Cependant, vous pourriez trouver son utilisation fastidieuse, principalement avec des opérations qui ne sont valides qu'avec une entrée. Dans ces cas, les combinateurs peuvent être utilisés pour gérer le contrôle du flux de manière plus modulaire.

Option possède une méthode préfaite appelée map(), un combinateur pour la simple mise en corrélation de Some -> Some et None -> None. Les appels de la méthode map() peuvent être chaînés ensemble pour plus de flexibilité.

Dans l'exemple suivant, process() remplace toutes les fonctions précédentes tout en restant compacte.

```
#![allow(dead_code)]

#[derive(Debug)] enum Food { Apple, Carrot, Potato }

#[derive(Debug)] struct Peeled(Food);
#[derive(Debug)] struct Chopped(Food);
#[derive(Debug)] struct Cooked(Food);


// Épluchage de la nourriture. S'il n'y en a pas, alors on renvoie `None`.
// Sinon, on renvoie la nourriture épluchée.
fn peel(food: Option<Food>) -> Option<Peeled> {
    match food {
        Some(food) => Some(Peeled(food)),
        None       => None,
    }
}

// Émincage de la nourriture. S'il n'y en a pas, alors on renvoie `None`.
// Sinon, on renvoie la nourriture émincée.
fn chop(peeled: Option<Peeled>) -> Option<Chopped> {
    match peeled {
        Some(Peeled(food)) => Some(Chopped(food)),
        None               => None,
    }
}

// Cuisson de la nourriture. Ici, nous utilisons `map()` au lieu de `match` pour la gestion des cas.
```

```
fn cook(chopped: Option<Chopped>) -> Option<Cooked> {
    chopped.map(|Chopped(food)| Cooked(food))
}

// Une fonction pour éplucher, émincer et cuire la nourriture dans une seule séquence.
// Nous chaînons plusieurs appels de `map()` pour simplifier le code.
fn process(food: Option<Food>) -> Option<Cooked> {
    food.map(|f| Peeled(f))
        .map(|Peeled(f)| Chopped(f))
        .map(|Chopped(f)| Cooked(f))
}

// On vérifie s'il y a de la nourritre ou pas avant d'essayer de la manger !
fn eat(food: Option<Cooked>) {
    match food {
        Some(food) => println!("Mmm. I love {:?}", food),
        None       => println!("Oh no! It wasn't edible."),
    }
}

fn main() {
    let apple = Some(Food::Apple);
    let carrot = Some(Food::Carrot);
    let potato = None;

    let cooked_apple = cook(chop(peel(apple)));
    let cooked_carrot = cook(chop(peel(carrot)));
    // Vous remarquerez que `process()` est bien plus lisible.
    let cooked_potato = process(potato);

    eat(cooked_apple);
    eat(cooked_carrot);
    eat(cooked_potato);
}
```

16-2-2 - Les combinateurs: and_then

map() a été décrite comme étant un moyen de chaîner les directives pour simplifier les déclarations match. Cependant, utiliser map() sur une fonction qui renvoie déjà une instance de Option<T> risque d'imbriquer le résultat dans une autre instance Option<Option<T>>. Chaîner des appels peut alors prêter à confusion. C'est là où un autre combinateur nommé and_then(), connu dans d'autres langages sous le nom de flatmap, entre en jeu.

and_then() appelle la fonction passée en entrée avec la valeur imbriquée et renvoie le résultat. Si le conteneur Option vaut None, alors elle renvoie None à la place.

Dans l'exemple suivant, cookable_v2() renvoie une instance de Option<Food>. Utiliser la méthode map() au lieu de and_then() donnerait une instance imbriquée Option<Option<Food>>, qui est un type invalide pour la fonction eat().

```
#![allow(dead_code)]

#[derive(Debug)] enum Food { CordonBleu, Steak, Sushi }
#[derive(Debug)] enum Day { Monday, Tuesday, Wednesday }

// Nous n'avons pas les ingrédients pour faire des sushis.
fn have_ingredients(food: Food) -> Option<Food> {
    match food {
        Food::Sushi => None,
        _           => Some(food),
    }
}

// Nous avons la recette de tous les mets sauf celle du Cordon Bleu.
fn have_recipe(food: Food) -> Option<Food> {
    match food {
        Food::CordonBleu => None,
        _                => Some(food),
    }
```

```
}


// Pour faire un plat, nous avons besoin de deux ingrédients et d'une recette.
// Nous pouvons représenter la logique avec une chaîne de `match`s:
fn cookable_v1(food: Food) -> Option<Food> {
    match have_ingredients(food) {
        None       => None,
        Some(food) => match have_recipe(food) {
            None       => None,
            Some(food) => Some(food),
        },
    }
}

// On peut rendre plus compacte cette implémentation en utilisant `and_then()`:
fn cookable_v2(food: Food) -> Option<Food> {
    have_ingredients(food).and_then(have_recipe)
}

fn eat(food: Food, day: Day) {
    match cookable_v2(food) {
        Some(food) => println!("Yay! On {:?} we get to eat {:?}.", day, food),
        None       => println!("Oh no. We don't get to eat on {:?}?", day),
    }
}

fn main() {
    let (cordon_bleu, steak, sushi) = (Food::CordonBleu, Food::Steak, Food::Sushi);

    eat(cordon_bleu, Day::Monday);
    eat(steak, Day::Tuesday);
    eat(sushi, Day::Wednesday);
}
```

Voir aussi

Les closures, **Option** et **Option::and_then()**.

16-3 - L'enum Result

Result est une version plus élaborée de l'enum Option qui conçoit une potentielle erreur plutôt qu'une potentielle *absence*.

Autrement dit, Result<T, E> pourrait adopter l'un de ces deux états :

- Ok<T> : Un élément T a été trouvé ;
- Err<E> : Une erreur a été trouvée avec l'élément E.

Par convention, la valeur de retour attendue est Ok jusqu'à la preuve du contraire (i.e. qu'une erreur (Err) est survenue).

Tout comme Option, Result possède de nombreuses méthodes associées à elle. unwrap(), par exemple, fournit l'élément T sinon déclenche panic. Pour la gestion des cas, Result possède nombre de combinateurs en commun avec Option.

En travaillant avec Rust, vous rencontrerez probablement des méthodes renvoyant le type Result, telles que la méthode **parse()**. Il n'est pas toujours possible de convertir une chaîne de caractères dans un autre type, donc parse() renvoie une instance de Result indiquant les potentielles erreurs.

Voyons ce qu'il se passe lorsque nous parvenons à convertir une chaîne de caractères et lorsque ce n'est pas le cas :

```
fn double_number(number_str: &str) -> i32 {
    // Essayons d'utiliser la méthode `unwrap` pour récupérer le nombre.
    // Va-t-elle nous mordre ?
    2 * number_str.parse::<i32>().unwrap()
}

fn main() {
    let twenty = double_number("10");
    println!("double is {}", twenty);

     let tt = double_number("t");
     println!("double is {}", tt);
 }
```

Dans le cas où nous ne parvenons pas à la convertir, parse() nous laisse avec l'erreur sur laquelle unwrap a paniqué. Vous noterez également que le message d'erreur affiché par panic est assez désagréable.

Pour améliorer la qualité de notre message d'erreur, nous devrions être plus rigoureux quant à la valeur de retour et envisager de gérer explicitement l'erreur.

16-3-1 - La méthode map pour Result

Nous avons noté dans l'exemple précédent que le message d'erreur affiché lorsque le programme "panique" ne nous était pas d'une grande aide. Pour éviter cela, nous devons être plus précis concernant le type de renvoi. Ici, l'élément est de type i32. Pour déterminer le type de Err, jetons un oeil à la documentation de la méthode **parse()**, qui est implémentée avec le trait **FromStr** pour le type **i32**. Dans un résultat, le type de Err est **ParseIntError**.

Dans l'exemple ci-dessous, utiliser directement match mène à coder quelque chose de relativement lourd. Heureusement, la méthode map de Option est l'un de ces nombreux combinateurs ont également été implémentés pour Result. **enum.Result** en tient une liste complète.

```
use std::num::ParseIntError;

// Maintenant que le type de renvoi a été réécrit, nous utilisons le pattern matching
// sans la méthode `unwrap()`.
fn double_number(number_str: &str) -> Result<i32, ParseIntError> {
    match number_str.parse::<i32>() {
        Ok(n)  => Ok(2 * n),
        Err(e) => Err(e),
    }
 }


 // Tout comme avec `Option`, nous pouvons utiliser les combinateurs tels que `map()`.
 // Cette fonction possède le même fonctionnement que celle ci-dessus et se lit comme suit:
 // Modifie n si la valeur est valide, sinon renvoie une erreur.
 fn double_number_map(number_str: &str) -> Result<i32, ParseIntError> {
     number_str.parse::<i32>().map(|n| 2 * n)
 }

 fn print(result: Result<i32, ParseIntError>) {
     match result {
         Ok(n)  => println!("n is {}", n),
         Err(e) => println!("Error: {}", e),
     }
 }

 fn main() {
     // Ceci fournit toujours une réponse valable.
     let twenty = double_number("10");
     print(twenty);

     // Ce qui suit fournit désormais un message d'erreur plus intelligible.
     let tt = double_number_map("t");
     print(tt);
 }
```

16-3-2 - Les alias de Result

Quid lorsque nous souhaitons réutiliser un type de Result bien précis ? Rappelez-vous que Rust nous permet de créer des **alias**. Nous pouvons alors aisément en définir un pour le type de Result en question.

À l'échelle d'un module, la création d'alias peut être salvatrice. Les erreurs pouvant être trouvées dans un module pécis ont souvent le même type (wrappé par Err), donc un seul alias peut définir *l'intégralité* des Results associés. C'est tellement utile que la bibliothèque standard en fournit un: io::Result !

Voici un petit exemple pour présenter la syntaxe :

```
use std::num::ParseIntError;

// On définit un alias générique pour un type de `Result` avec le type d'erreur
// `ParseIntError`.
type AliasedResult<T> = Result<T, ParseIntError>;

// On utilise l'alias ci-dessus pour faire référence à notre
// `Result`.
fn double_number(number_str: &str) -> AliasedResult<i32> {
    number_str.parse::<i32>().map(|n| 2 * n)
}

// Ici, l'alias nous permet encore d'épargner de l'espace.
fn print(result: AliasedResult<i32>) {
    match result {
        Ok(n)  => println!("n is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    print(double_number("10"));
    print(double_number("t"));
}
```

Voir aussi

Result et **io::Result**.

16-4 - Multiples types d'erreur

Les exemples précédents ont toujours été très basiques; Les (instances de) Result interagissent avec d'autres (instances de) Result et les Options interagissent avec d'autres Options.

Parfois une instance d'Option a besoin d'interagir avec un Result ou encore un Result<T, Error1> devant interagir avec un Result<T, Error2>. Dans ces cas, nous souhaitons gérer nos différents types de manière à pouvoir interagir simplement avec eux.

Dans le code suivant, deux instances de la méthode unwrap() génèrent deux types d'erreur différents. Vec::first renvoie une instance de Option, alors que parse::<i32> renvoie une instance de Result<i32, ParseIntError> :

```
fn double_first(vec: Vec<&str>) -> i32 {
    let first = vec.first().unwrap(); // Génère la première erreur.
    2 * first.parse::<i32>().unwrap() // Génère la seconde erreur.
}

fn main() {
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

```

```
    println!("The first doubled is {}", double_first(empty));
    // Erreur 1: Le vecteur passé en entrée est vide.

    println!("The first doubled is {}", double_first(strings));
    // Erreur 2: L'élément ne peut pas être converti en nombre.
}
```

En utilisant notre connaissance des combinateurs, nous pouvons réécrire ce qu'il y a au-dessus pour gérer explicitement les erreurs. Puisque deux types différents peuvent être rencontrés, nous nous devons de les convertir en un type commun tel que String.

Pour ce faire, nous convertissons les instances d'Option et de Result en Results puis nous convertissons leurs erreurs respectives sous le même type :

```
// Nous utiliserons `String` en tant que type d'erreur.
type Result<T> = std::result::Result<T, String>;

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       // On convertit l'`Option` en un `Result` s'il y a une valeur.
       // Autrement, on fournit une instance `Err` contenant la `String`.
       .ok_or("Please use a vector with at least one element.".to_owned())
       .and_then(|s| s.parse::<i32>()
                        // On convertit n'importe quelle erreur, générée par `parse`,
                        // en `String`.
                        .map_err(|e| e.to_string())
                        // `Result<T, String>` est le nouveau type de valeur,
                        // et nous pouvons doubler le nombre se trouvant dans le conteneur.
                        .map(|i| 2 * i))
}

fn print(result: Result<i32>) {
     match result {
         Ok(n)  => println!("The first doubled is {}", n),
         Err(e) => println!("Error: {}", e),
     }
}

fn main() {
     let empty = vec![];
     let strings = vec!["tofu", "93", "18"];

     print(double_first(empty));
     print(double_first(strings));
}
```

Dans la prochaine section, nous allons voir une méthode pour la gestion explicite de ces erreurs.

Voir aussi

Option::ok_or, **Result::map_err**.

16-4-1 - Retour prématuré

Dans l'exemple précédent, nous gérions explicitement les erreurs en utilisant les combinateurs. Une autre manière de répondre à cette analyse est d'utiliser une série de match et des *retours prématurés*.

Autrement dit, nous pouvons simplement mettre fin à l'exécution de la fonction et renvoyer l'erreur, s'il y en a une. Pour certains, cette manière de faire est plus simple à lire et écrire. Voici une nouvelle version de l'exemple précédent, réécrit en utilisant les retours prématurés :

```
// On utilise `String` comme type d'erreur.
type Result<T> = std::result::Result<T, String>;
```

```
fn double_first(vec: Vec<&str>) -> Result<i32> {
    // On convertit l'`Option` en un `Result` s'il y a une valeur.
    // Autrement, on fournit une instance `Err` contenant la `String`.
    let first = match vec.first() {
        Some(first) => first,
        None => return Err("Please use a vector with at least one element.".to_owned()),
     };

     // On double le nombre dans le conteneur si `parse` fonctionne
     // correctement.
     // Sinon, on convertit n'importe quelle erreur, générée par `parse`,
     // en `String`.
     match first.parse::<i32>() {
         Ok(i) => Ok(2 * i),
         Err(e) => Err(e.to_string()),
     }
}

fn print(result: Result<i32>) {
     match result {
         Ok(n) => println!("The first doubled is {}", n),
         Err(e) => println!("Error: {}", e),
     }
}

fn main() {
     let empty = vec![];
     let strings = vec!["tofu", "93", "18"];

     print(double_first(empty));
     print(double_first(strings));
}
```

À ce niveau, nous avons appris à gérer explicitement les erreurs en utilisant les combinateurs et les retours prématurés. Bien que, généralement, nous souhaitons éviter un plantage, la gestion explicite de toutes nos erreurs peut s'avérer relativement lourde.

Dans la section suivante, nous introduirons la macro try! pour couvrir les cas où nous souhaitons simplement utiliser unwrap() sans faire planter le programme.

16-4-2 - Introduction à try!

Parfois nous voulons la simplicité de unwrap() sans la possibilité de faire planter le programme. Jusqu'ici, unwrap() nous a dévié de ce que nous voulions vraiment : *récupérer la variable*. C'est exactement le but de try! que de régler ce souci.

Une fois l'instance Err trouvée, il y a deux actions possibles :

1 panic! que nous avons choisi d'éviter, si possible, avec try! ;
2 return parce qu'une Err ne peut être traitée.

La macro try! est *presque* (1) équivalent à la méthode unwrap() mais effectue un renvoi prématuré au lieu de planter lorsqu'un conteneur Err est récupéré. Voyons comment nous pouvons simplifier l'exemple précédent qui utilisait les combinateurs :

```
// On utilise une `String` comme type d'erreur.
type Result<T> = std::result::Result<T, String>;

fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = try!(vec.first().ok_or(
        "Please use a vector with at least one element.".to_owned(),
    ));

```

```
    let value = try!(first.parse::<i32>().map_err(|e| e.to_string()));

     Ok(2 * value)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n) => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(empty));
    print(double_first(strings));
}
```

Notez que, jusqu'ici, nous avons utilisé les Strings pour les erreurs. Cependant, elles sont quelque peu limitées en tant que type d'erreur. Dans la prochaine section, nous apprendrons à créer des erreurs plus structurées, plus riches, en définissant leur propre type.

Voir aussi

Result et **io::Result**.

16-5 - Définition d'un type d'erreur

Rust nous permet de définir nos propres types d'erreur. En général, un « bon » type d'erreur :

- représente différentes erreurs avec le même type ;
- présente un message d'erreur intelligible pour l'utilisateur ;
- est facilement comparable aux autres types ;
    - Bien : Err(EmptyVec),
    - Pas bien : Err("Please use a vector with at least one element".to_owned()) ;
- peut supporter l'ajout d'informations à propos de l'erreur ;
    - Bien : Err(BadChar(c, position)),
    - Pas bien : Err("+ cannot be used here".to_owned()).

Notez qu'une String (que nous utilisions jusqu'ici) remplit les deux premiers critères, mais pas les deux derniers. Cela rend la création d'erreurs, simplement en utilisant String, verbeuse et difficile à maintenir. Il ne devrait pas être nécessaire de polluer la logique du code avec le formattage des chaînes de caractères pour avoir un affichage intelligible.

```
use std::num::ParseIntError;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug)]
// On définit nos propres types d'erreur. Ces derniers peuvent être personnalisés pour
// notre gestion des cas.
// Maintenant nous serons capables d'écrire nos propres erreurs et nous reporter
 // à leur implémentation.
 enum DoubleError {
    // Il n'est pas nécessaire de collecter plus d'informations
    // pour détailler cette erreur.
    EmptyVec,
```

```
    // Nous allons nous reporter à l'implémentation couvrant l'erreur de conversion pour ce type.
    // Soumettre des informations supplémentaires nécéssite l'ajout de données pour le type.
    Parse(ParseIntError),
}

// La génération d'une erreur fait abstraction de la manière dont elle est affichée,
// nul besoin de s'occuper de la mécanique sous-jacente.
//
// Notez que nous ne stockons aucune information supplémentaire à propos de l'erreur.
// Cela signifie que nous ne pouvons préciser quelle chaîne de caractères n'a pas pu être
// convertie, sans modifier nos types pour apporter cette information.
impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            DoubleError::EmptyVec => write!(f, "please use a vector with at least one element"),
            // Ceci est un wrapper, donc référrez-vous à l'implémentation de `fmt` respective
            // à chaque type.
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    vec.first()
       // On remplace l'erreur par notre nouveau type.
       .ok_or(DoubleError::EmptyVec)
       .and_then(|s| s.parse::<i32>()
            // On met également à jour le nouveau type d'erreur ici.
            .map_err(DoubleError::Parse)
            .map(|i| 2 * i))
}

fn print(result: Result<i32>) {
    match result {
        Ok(n) => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

Voir aussi

Result et **io::Result**.

16-6 - D'autres cas d'utilisation de try!

Vous avez remarqué, dans l'exemple précédent, que notre réaction immédiate à l'appel de parse « était » de passer l'erreur de la bibliothèque dans notre nouveau type :

```
.and_then(|s| s.parse::<i32>()
    .map_err(DoubleError::Parse)
```

Puisque c'est une opération plutôt commune, il ne serait pas inutile qu'elle soit élidée. Hélas, and_then n'étant pas suffisamment flexible pour cela, ce n'est pas possible. Nous pouvons, à la place, utiliser try!.

La macro try! a été précédemment présentée comme permettrant la récupération de la ressource (unwrap) ou le renvoi prématuré, si une erreur survient (return Err(err)). C'est plus ou moins vrai. En réalité, elle utilise soit unwrap soit return Err(From::from(err)). Puisque From::from est un utilitaire permettant la conversion entre différents types, cela signifie que si vous utilisez try! où l'erreur peut être convertie au type de retour, elle le sera automatiquement.

Ici, nous réécrivons l'exemple précédent en utilisant try!. Résultat, la méthode map_err disparaîtra lorsque From::from sera implémenté pour notre type d'erreur :

```rust
use std::num::ParseIntError;
use std::fmt;

type Result<T> = std::result::Result<T, DoubleError>;

#[derive(Debug)]
enum DoubleError {
    EmptyVec,
    Parse(ParseIntError),
}

// On implémente la conversion du type `ParseIntError` au type `DoubleError`.
// La conversion sera automatiquement appelée par `try!` si une instance `ParseIntError`
// doit être convertie en `DoubleError`.
impl From<ParseIntError> for DoubleError {
    fn from(err: ParseIntError) -> DoubleError {
        DoubleError::Parse(err)
    }
}

impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            DoubleError::EmptyVec =>
                write!(f, "please use a vector with at least one element"),
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

// La même structure qu'avant mais, plutôt que de chaîner les instances `Result`
// et `Option` tout du long, nous utilisons `try!` pour récupérer immédiatement
// la valeur contenue.
fn double_first(vec: Vec<&str>) -> Result<i32> {
    // Convertit toujours en `Result` tout en renseignant comment convertir
    // un `None`.
    let first = try!(vec.first().ok_or(DoubleError::EmptyVec));
    let parsed = try!(first.parse::<i32>());

    Ok(2 * parsed)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

C'est effectivement plus acceptable. Comparé à l'original panic, en remplaçant les appels de unwrap par try! nous conservons une utilisation assez familière, à l'exception que try! renvoie les types dans un conteneur Result, rendant leur déstructuration plus abstraite.

Notez toutefois que vous ne devriez pas systématiquement gérer les erreurs de cette manière pour remplacer les appels de unwrap. Cette méthode de gestion d'erreur a triplé le nombre de lignes de code et ne peut pas être considéré comme « simple » (même si la taille du code n'est pas énorme).

En effet, modifier une bibliothèque de 1000 lignes pour remplacer des appels de unwrap et établir une gestion des erreurs plus « propre » pourrait être faisable en une centaine de lignes supplémentaires. En revanche, le recyclage nécessaire en aval ne serait pas évident.

Nombreuses sont les bibliothèques qui pourraient s'en sortir en implémentant seulement Display et ajouter From comme base pour la gestion. Cependant, pour des bibliothèques plus importantes, l'implémentation de la gestion des erreurs peut répondre à des besoins plus spécifiques.

Voir aussi

From::from et **try!**.

16-7 - Boxing des erreurs

En implémentant Display et Form pour notre type d'erreur, nous avons usé de *presque* tous les outils dédiés à la gestion d'erreur de la bibliothèque standard. Nous avons cependant oublié quelque chose : la capacité à simplement Box notre type.

La bibliothèque standard convertit n'importe quel type qui implémente le trait Error et sera pris en charge par le type Box<Error>, via From. Pour l'utilisateur d'une bibliothèque, ceci permet aisément une manoeuvre de ce genre :

```
fn foo(...) -> Result<T, Box<Error>> { ... }
```

Un utilisateur peut utiliser nombre de bibliothèques externes, chacune fournissant leurs propres types d'erreur. Pour définir un type de Result<T, E> valide, l'utilisateur a plusieurs options :

- Définir un nouveau wrapper englobant les types d'erreur de la bibliothèque ;
- Convertir les types d'erreur en String ou vers un autre type intermédiaire ;
- Box les types dans Box<Error>.

Le "boxing" du type d'erreur est un choix plutôt habituel. Le problème est que le type de l'erreur sous-jacente est connu à l'exécution et n'est pas **déterminé statiquement**. Comme mentionné plus haut, tout ce qu'il y a à faire c'est d'implémenter le trait Error :

```
trait Error: Debug + Display {
    fn description(&self) -> &str;
    fn cause(&self) -> Option<&Error>;
}
```

Avec cette implémentation, jetons un oeil à notre exemple récemment présenté. Notez qu'il est tout aussi fonctionnel avec le type Box<Error> qu'avec DoubleError :

```
use std::error;
use std::fmt;
use std::num::ParseIntError;

// On modifie l'alias pour ajouter `Box<error::Error>`.
type Result<T> = std::result::Result<T, Box<error::Error>>;

```

```
#[derive(Debug)]
enum DoubleError {
    EmptyVec,
    Parse(ParseIntError),
}

impl From<ParseIntError> for DoubleError {
    fn from(err: ParseIntError) -> DoubleError {
        DoubleError::Parse(err)
    }
}

impl fmt::Display for DoubleError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            DoubleError::EmptyVec =>
                write!(f, "please use a vector with at least one element"),
            DoubleError::Parse(ref e) => e.fmt(f),
        }
    }
}

impl error::Error for DoubleError {
    fn description(&self) -> &str {
        match *self {
            // Courte description de l'erreur.
            // Vous n'êtes pas obligé de renseigner la même description que
            // pour `Display`.
            DoubleError::EmptyVec => "empty vectors not allowed",
            // Ceci implémente déjà `Error`, on se reporte à sa propre implémentation.
            DoubleError::Parse(ref e) => e.description(),
        }
    }

    fn cause(&self) -> Option<&error::Error> {
        match *self {
            // Pas de cause (i.e. pas d'autre erreur)
            // sous-jacente au déclenchement de cette erreur, donc on renvoie `None`.
            DoubleError::EmptyVec => None,
            // La cause est l'implémentation sous-jacente du type d'erreur.
            // Il (le type) est implicitement casté en `&error::Error`.

// Ca fonctionne parce que le type en question a déjà implémenté le trait `Error`.
            DoubleError::Parse(ref e) => Some(e),
        }
    }
}

fn double_first(vec: Vec<&str>) -> Result<i32> {
    let first = try!(vec.first().ok_or(DoubleError::EmptyVec));
    let parsed = try!(first.parse::<i32>());

    Ok(2 * parsed)
}

fn print(result: Result<i32>) {
    match result {
        Ok(n)  => println!("The first doubled is {}", n),
        Err(e) => println!("Error: {}", e),
    }
}

fn main() {
    let numbers = vec!["93", "18"];
    let empty = vec![];
    let strings = vec!["tofu", "93", "18"];

    print(double_first(numbers));
    print(double_first(empty));
    print(double_first(strings));
}
```

Voir aussi

Distribution dynamique et **le trait Error**.

17 - Les types de la bibliothèque standard

La bibliothèque standard fournit de nombreux types complexes qui étendent drastiquement l'utilisation des primitifs. Voici certains d'entre-eux :

- Les chaînes de caractères redimensionnables String : "hello world" ;
- Les vecteurs redimensionnables : [1, 2, 3] ;
- Les types optionnels : Option<i32> ;
- Les types dédiés à la gestion des erreurs : Result<i32, i32> ;
- Les pointeurs alloués dans le tas : Box<i32>.

Voir aussi

Les primitifs et **la bibliothèque standard**.

17-1 - Les Box, la pile et le tas

En Rust, toutes les valeurs sont allouées dans la pile, par défaut. Les valeurs peuvent être *boxées* (i.e. allouées dans le tas) en créant une instance de Box<T>. Une "box" est un pointeur intelligent sur une ressource de type T allouée dans le tas. Lorsqu'une box sort du contexte, son destructeur est appelé, l'objet à charge est détruit et la mémoire du tas libérée.

Les valeurs « boxées » peuvent être déréférencées en utilisant l'opérateur * ; Ceci supprime un **niveau d'indirection**.

```rust
use std::mem;

#[derive(Clone, Copy)]
struct Point {
    x: f64,
    y: f64,
}

#[allow(dead_code)]
struct Rectangle {
    p1: Point,
    p2: Point,
}

fn origin() -> Point {
    Point { x: 0.0, y: 0.0 }
}

fn boxed_origin() -> Box<Point> {
    // Alloue cette instance de `Point` dans le tas et renvoie un pointeur
    // sur cette dernière.
    Box::new(Point { x: 0.0, y: 0.0 })
}

fn main() {
    // (Ici le typage est superflu)
    // Variables allouées dans la pile.
    let point: Point = origin();
    let rectangle: Rectangle = Rectangle {
        p1: origin(),
        p2: Point { x: 3.0, y: 4.0 }
    };

    // Rectangle alloué dans le tas.
```

```
    let boxed_rectangle: Box<Rectangle> = Box::new(Rectangle {
        p1: origin(),
        p2: origin()
    });

    // Le résultat des fonctions peut être boxé également.
    let boxed_point: Box<Point> = Box::new(origin());

    // Double indirection
    let box_in_a_box: Box<Box<Point>> = Box::new(boxed_origin());

    println!("Point occupies {} bytes in the stack",
             mem::size_of_val(&point));
    println!("Rectangle occupies {} bytes in the stack",
             mem::size_of_val(&rectangle));

    // La taille du pointeur est égale à la taille de la Box.
    println!("Boxed point occupies {} bytes in the stack",
             mem::size_of_val(&boxed_point));
    println!("Boxed rectangle occupies {} bytes in the stack",
             mem::size_of_val(&boxed_rectangle));
    println!("Boxed box occupies {} bytes in the stack",
             mem::size_of_val(&box_in_a_box));

    // La ressource contenue dans `boxed_point` est copiée dans
    // `unboxed_point`.
    let unboxed_point: Point = *boxed_point;
    println!("Unboxed point occupies {} bytes in the stack",
             mem::size_of_val(&unboxed_point));
}
```

17-2 - Les vecteurs

Les vecteurs sont des tableaux redimensionnables. Tout comme les slices, leur taille n'est pas connue à la compilation mais ils peuvent être agrandis ou tronqués au cours de l'exécution. Un vecteur est représenté par trois (3) mots : un pointeur sur la ressource, sa taille et sa capacité. La capacité indique la quantité de mémoire réservée au vecteur. La taille peut augmenter à volonté, tant qu'elle est inférieure à la capacité. Lorsqu'il est nécessaire de franchir cette limite, le vecteur est réalloué avec une capacité plus importante.

```
fn main() {
    // Les éléments des itérateurs peuvent être collectés et
    // ajoutés dans un vecteur.
    let collected_iterator: Vec<i32> = (0..10).collect();
    println!("Collected (0..10) into: {:?}", collected_iterator);

    // La macro `vec!` peut être utilisée pour initialiser un vecteur.
    let mut xs = vec![1i32, 2, 3];
    println!("Initial vector: {:?}", xs);

     // On ajoute un nouvel élément à la fin du vecteur.
     println!("Push 4 into the vector");
     xs.push(4);
     println!("Vector: {:?}", xs);

     // Erreur! Les vecteurs immuables ne peuvent pas être
     // agrandis.
     // collected_iterator.push(0);
     // FIXME ^ Commentez/décommentez cette ligne

     // La méthode `len` renvoie la taille actuelle du vecteur.
     println!("Vector size: {}", xs.len());

     // L'indexation peut être faite à l'aide des "[]" (l'indexaction débute à 0).
     println!("Second element: {}", xs[1]);

     // `pop` supprime le dernier élément du vecteur et le renvoie.
     println!("Pop last element: {:?}", xs.pop());

     // Une indexaction hors des capacités du vecteur
```

```
    // mène à un plantage du programme.
    println!("Fourth element: {}", xs[3]);
}
```

Les méthodes rattachées à la structure Vec peuvent être trouvées au sein du module **std::vec**.

17-3 - Les chaînes de caractères

Il y a deux types de chaînes de caractères en Rust : String et &str.

Une instance de String est stockée en tant que vecteur d'octets (Vec<u8>) mais garantit de toujours fournir une séquence valide encodée en UTF-8. String est alloué dans le tas, redimensionnable et non-nul.

&str est une slice (&[u8]) qui pointe toujours sur une séquence UTF-8 valide et peut être utilisée comme une vue sur une String. Tout comme &[T] est une vue sur une instance Vec<T>.

```
fn main() {
    // (Le typage est optionnel)
    // Une référence d'une chaîne de caractères immuable.
    let pangram: &'static str = "the quick brown fox jumps over the lazy dog";
    println!("Pangram: {}", pangram);

    // On itère sur les mots dans le sens inverse, aucune nouvelle instance
    // n'est créée.
    println!("Words in reverse");
     for word in pangram.split_whitespace().rev() {
         println!("> {}", word);
     }

     // On copie les caractères dans un vecteur, les trie et supprime
     // les occurrences multiples.
     let mut chars: Vec<char> = pangram.chars().collect();
     chars.sort();
     chars.dedup();

     // On créé une instance de `String` vide et mutable.
     let mut string = String::new();
     for c in chars {
         // On ajoute un caractère à la fin de la chaîne.
         string.push(c);
         // On ajoute une nouvelle chaîne à la fin de la chaîne initiale.
         string.push_str(", ");
     }

     // La chaîne tronquée est une slice de la chaîne originale, il n'y a
     // pas de nouvelle allocation.
     let chars_to_trim: &[char] = &[' ', ','];
     let trimmed_str: &str = string.trim_matches(chars_to_trim);
     println!("Used characters: {}", trimmed_str);

     // Chaîne allouée dans le tas.
     let alice = String::from("I like dogs");
     // Nouvelle allocation mémoire et stockage de la chaîne modifiée
     // à cet endroit.
     let bob: String = alice.replace("dog", "cat");

     println!("Alice says: {}", alice);
     println!("Bob says: {}", bob);
}
```

Les méthodes rattachées à str/String peuvent être trouvées dans les modules **std::str** et **std::string**.

17-4 - L'énumération Option

Il est parfois désirable de rattraper les erreurs provenants de différentes parties du programme plutôt que d'appeler panic! ; pour ce faire, l'enum Option prend le relais.

L'enum Option<T> possède deux variantes :

1 None, pour signaler une erreur ou l'absence d'une valeur, et
2 Some(value), un tuple qui enveloppe, contient une valeur de type T.

```
// Une division entre deux entiers qui ne plante pas.
fn checked_division(dividend: i32, divisor: i32) -> Option<i32> {
    if divisor == 0 {
        // L'échec est représenté par la variante `None`.
        None
    } else {
        // Le résultat est enveloppé dans une instance `Some`.
        Some(dividend / divisor)
    }
}

// Cette fonction gère une division qui peut ne pas fonctionner.
fn try_division(dividend: i32, divisor: i32) {
    // Les valeurs d'`Option` peuvent être matchées, tout comme les autres enums.
    match checked_division(dividend, divisor) {
        None => println!("{} / {} failed!", dividend, divisor),
        Some(quotient) => {
            println!("{} / {} = {}", dividend, divisor, quotient)
        },
    }
}

fn main() {
    try_division(4, 2);
    try_division(1, 0);

    // L'assignation de `None` à une variable nécessite de typer cette dernière.
    let none: Option<i32> = None;
    let _equivalent_none = None::<i32>;

    let optional_float = Some(0f32);

    // "dé-wrapper" une instance de `Some` va extraire la valeur contenue.
    println!("{:?} unwraps to {:?}", optional_float, optional_float.unwrap());

    // Tenter d'utiliser `unwrap` sur la variante `None` fera planter le programme.
    println!("{:?} unwraps to {:?}", none, none.unwrap());
}
```

17-5 - L'énumération Result

Nous avons vu que l'enum Option peut être utilisée en tant que valeur de retour depuis les fonctions pouvant échouer, où None peut être renvoyé pour indiquer un échec. Il est parfois important d'expliquer *pourquoi* une opération a échoué. Pour ce faire, nous pouvons utiliser Result.

Result<T, E> possède deux variantes :

1 Ok(valeur) qui signale que l'opération s'est correctement déroulée et enveloppe la valeur renvoyée par l'opération (valeur est de type T);
2 Err(pourquoi) qui signale que l'opération a échoué et enveloppe le pourquoi, qui (espérons-le) nous renseigne sur la cause de l'échec (pourquoi est de type E).

```
mod checked {
```

```
    // Les "erreurs" mathématiques que nous voulons gérer.
    #[derive(Debug)]
    pub enum MathError {
        DivisionByZero,
        NonPositiveLogarithm,
        NegativeSquareRoot,
    }

     pub type MathResult = Result<f64, MathError>;

     pub fn div(x: f64, y: f64) -> MathResult {
         if y == 0.0 {
             // Cette opération échouerait, nous wrappons l'erreur dans une instance
             // `Err` à la place.
             Err(MathError::DivisionByZero)
         } else {
             // Cette opération est valide, nous renvoyons le résultat wrappé dans `Ok`.
             Ok(x / y)
         }
     }

     pub fn sqrt(x: f64) -> MathResult {
         if x < 0.0 {
             Err(MathError::NegativeSquareRoot)
         } else {
             Ok(x.sqrt())
         }
     }

     pub fn ln(x: f64) -> MathResult {
         if x <= 0.0 {
             Err(MathError::NonPositiveLogarithm)
         } else {
             Ok(x.ln())
         }
     }
}

// `op(x, y)` === `sqrt(ln(x / y))`
fn op(x: f64, y: f64) -> f64 {
    // Ceci est une pyramide de match à trois niveaux !
    match checked::div(x, y) {
        Err(why) => panic!("{:?}", why),
        Ok(ratio) => match checked::ln(ratio) {
            Err(why) => panic!("{:?}", why),
            Ok(ln) => match checked::sqrt(ln) {
                Err(why) => panic!("{:?}", why),
                Ok(sqrt) => sqrt,
            },
        },
    }
}

fn main() {
    // Cette opération va-t-elle échouer ?
    println!("{}", op(1.0, 10.0));
}
```

17-5-1 - La macro try!

Chaîner les résultats en utilisant match peut être très chaotique ; heureusement, la macro try! peut être utilisée pour soigner l'écriture. La macro try! étend une expression de match où la branche Err(err) étend un retour prématuré (return Err(err)) et la branche Ok(ok) étend une expression ok et fournit la ressource.

```
mod checked {
    #[derive(Debug)]
    enum MathError {
        DivisionByZero,
        NegativeLogarithm,
```

```
        NegativeSquareRoot,
    }

    type MathResult = Result<f64, MathError>;

    fn div(x: f64, y: f64) -> MathResult {
        if y == 0.0 {
            Err(MathError::DivisionByZero)
        } else {
            Ok(x / y)
        }
    }

    fn sqrt(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeSquareRoot)
        } else {
            Ok(x.sqrt())
        }
    }

    fn ln(x: f64) -> MathResult {
        if x < 0.0 {
            Err(MathError::NegativeLogarithm)
        } else {
            Ok(x.ln())
        }
    }

    // Fonction intermédiaire.
    fn op_(x: f64, y: f64) -> MathResult {
        // Si la fonction `div` échoue, alors `DivisionByZero` sera renvoyée.
        let ratio = try!(div(x, y));

        // Si `ln` échoue, alors `NegativeLogarithm` sera renvoyée.
        let ln = try!(ln(ratio));

        sqrt(ln)
    }

    pub fn op(x: f64, y: f64) {
        match op_(x, y) {
            Err(why) => panic!(match why {
                MathError::NegativeLogarithm
                    => "logarithm of negative number",
                MathError::DivisionByZero
                    => "division by zero",
                MathError::NegativeSquareRoot
                    => "square root of negative number",
            }),
            Ok(value) => println!("{}", value),
        }
    }
}

fn main() {
    checked::op(1.0, 10.0);
}
```

N'hésitez pas à consulter la **documentation**, de nombreuses méthodes sont disponibles pour créer et gérer les Result.

17-6 - La macro panic!

La macro panic! peut être utilisée pour générer un plantage et dérouler la pile. Pendant le déroulement de la pile, l'exécution prendra soin de libérer toutes les ressources *possédées* par le fil d'exécution en appelant le destructeur de chaque objet.

Puisque nous interagissons avec nos programmes en n'utilisant qu'un seul fil d'exécution, panic! renverra un message d'erreur puis mettra un terme à l'exécution.

```
// Ré-implémentation de la division d'entiers (/).
fn division(dividend: i32, divisor: i32) -> i32 {
    if divisor == 0 {
        // La division par zéro fait planter le thread courant.
        panic!("division by zero");
    } else {
        dividend / divisor
    }
}


fn main() {
    // Entier alloué dans le tas.
    let _x = Box::new(0i32);

    // Cette opération va déclencher la procédure d'abandon.
    division(3, 0);

    println!("This point won't be reached!");

    // `_x` devrait être détruit à ce niveau.
}
```

Vérifions que la macro panic! ne cause aucune fuite mémoire.

```
$ rustc panic.rs && valgrind ./panic
==4401== Memcheck, a memory error detector
==4401== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4401== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==4401== Command: ./panic
==4401==
thread '<main>' panicked at 'division by zero', panic.rs:5
==4401==
==4401== HEAP SUMMARY:
==4401==     in use at exit: 0 bytes in 0 blocks
==4401==   total heap usage: 18 allocs, 18 frees, 1,648 bytes allocated
==4401==
==4401== All heap blocks were freed -- no leaks are possible
==4401==
==4401== For counts of detected and suppressed errors, rerun with: -v
==4401== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

17-7 - La structure HashMap

Là où les vecteurs stockent leurs valeurs en utilisant un index entier, les HashMaps stockent leurs valeurs en utilisant des clés. Les clés d'une HashMap peuvent être des booléens, des chaînes de caractères ou n'importe quel autre type qui implémente les traits Eq et Hash. Nous y reviendrons dans la section suivante.

Tout comme les vecteurs, les HashMap sont redimensionnables mais peuvent également se tronquer elles-mêmes lorsqu'elles atteignent la limite de leur capacité. Vous pouvez créer une HashMap avec une capacité donnée en utilisant HashMap::with_capacity(uint), ou utiliser HashMap::new() pour récupérer une instance avec une capacité initiale par défaut (recommandé).

```
use std::collections::HashMap;

fn call(number: &str) -> &str {
    match number {
        "798-1364" => "We're sorry, the call cannot be completed as dialed.
            Please hang up and try again.",
        "645-7689" => "Hello, this is Mr. Awesome's Pizza. My name is Fred.
            What can I get for you today?",
        _ => "Hi! Who is this again?"
```

```
    }
}

fn main() {
    let mut contacts = HashMap::new();

    contacts.insert("Daniel", "798-1364");
    contacts.insert("Ashley", "645-7689");
    contacts.insert("Katie", "435-8291");
    contacts.insert("Robert", "956-1745");

    // Prend une référence en entrée et renvoie un conteneur `Option<&V>`.
    match contacts.get(&"Daniel") {
        Some(&number) => println!("Calling Daniel: {}", call(number)),
        _ => println!("Don't have Daniel's number."),
    }


    // La méthode `HashMap::insert()` renvoie `None`
    // si la valeur insérée est nouvelle, sinon `Some(value)`.
    contacts.insert("Daniel", "164-6743");

    match contacts.get(&"Ashley") {
        Some(&number) => println!("Calling Ashley: {}", call(number)),
        _ => println!("Don't have Ashley's number."),
    }

    contacts.remove(&("Ashley"));

    // La méthode `HashMap::iter()` renvoie un itérateur qui fournit
    // les paires (&'a key, &'a value) dans un ordre arbitraire.
    for (contact, &number) in contacts.iter() {
        println!("Calling {}: {}", contact, call(number));
    }
}
```

Pour plus d"informations à propos du fonctionnement du hashage et des hash maps (parfois appelées hash tables), consultez **la page wikipédia dédiée aux Hash Tables**.

17-7-1 - Personnaliser les types de clé

N'importe quel type implémentant les traits Eq et Hash peuvent être une clé dans une HashMap. Ce qui inclut :

- Le type bool (Bien que peut utile puisqu'il n'y a que deux clés possibles);
- Le type int, uint et toutes les variantes de ces derniers ;
- String et &str (*note* : vous pouvez avoir une HashMap recevant en entrée des String et appeler la méthode .get() avec une &str).

Notez que f32 et f64 n'implémentent pas Hash, sûrement parce les erreurs de précision rendrait leur utilisation en tant que clé d'une hashmap poserait des soucis.

Toutes les classes représentant une collection implémentent Eq et Hash si le type qu'elles contiennent implémentent également ces deux traits. Par exemple, Vec<T> implémentera Hash si T l'implémente.

Vous pouvez facilement implémenter Eq et Hash pour un nouveau type avec cette seule ligne : #[derive(PartialEq, Hash)].

Le compilateur fera le reste. Si vous souhaitez avoir plus de contrôle sur les détails, vous pouvez implémenter Eq et/ou Hash vous-même. Ce guide ne couvre pas les implémentations spécifiques de Hash.

Pour tester l'utilisation d'une struct dans une HashMap, créons un simple système d'identification :

```
use std::collections::HashMap;
```

```rust

// `Eq` nécessite de dériver `PartialEq` sur le type.
#[derive(PartialEq, Eq, Hash)]
struct Account<'a>{
    username: &'a str,
    password: &'a str,
}

struct AccountInfo<'a>{
    name: &'a str,
    email: &'a str,
}

type Accounts<'a> = HashMap<Account<'a>, AccountInfo<'a>>;

fn try_logon<'a>(accounts: &Accounts<'a>,
        username: &'a str, password: &'a str){
    println!("Username: {}", username);
    println!("Password: {}", password);
    println!("Attempting logon...");

    let logon = Account {
        username: username,
        password: password,
    };

    match accounts.get(&logon) {
        Some(account_info) => {
            println!("Successful logon!");
            println!("Name: {}", account_info.name);
            println!("Email: {}", account_info.email);
        },
        _ => println!("Login failed!"),
    }
}

fn main(){
    let mut accounts: Accounts = HashMap::new();

    let account = Account {
        username: "j.everyman",
        password: "password123",
    };

    let account_info = AccountInfo {
        name: "John Everyman",
        email: "j.everyman@email.com",
    };

    accounts.insert(account, account_info);

    try_logon(&accounts, "j.everyman", "psasword123");

    try_logon(&accounts, "j.everyman", "password123");
}
```

17-7-2 - La structure HashSet

Voyez une HashSet comme une HashMap où nous nous soucions uniquement des clés (HashSet<T> est, en réalité, simplement un wrapper de HashMap<T, ()>).

Vous pourriez vous demander "Quel est le but ? Je pourrais simplement stocker mes clés dans un Vec".

La fonctionnalité unique de HashSet est qu'elle garantit l'inexistance d'éléments dupliqués. C'est le contrat que n'importe quel ensemble remplit. HashSet n'est qu'une implémentation (voir aussi : **BTreeSet**).

Si vous ajoutez une valeur déjà présente dans l'instance HashSet, (i.e. la nouvelle valeur est égale à l'existante et ont toutes deux le même hash), alors la nouvelle valeur remplacera l'ancienne.

C'est pratique lorsque vous ne souhaitez jamais plus d'une occurrence de quelque chose ou lorsque vous voulez savoir si vous possédez déjà quelque chose. Mais les ensembles peuvent faire bien plus que cela.

Les ensembles ont quatre (4) opérations inhérentes (chacune renvoie un itérateur) :

1. union : Récupère tous les éléments dans les deux ensembles ;
2. difference : Récupère tous les éléments qui sont dans le premier ensemble mais pas dans le second ;
3. intersection : Récupère uniquement les éléments présents dans les *deux* ensembles ;
4. symmetric_difference : Récupère tous éléments qui sont dans le premier ou second ensemble mais *pas* les deux.

Essayons tout cela dans l'exemple suivant.

```
use std::collections::HashSet;

fn main() {
    let mut a: HashSet<i32> = vec!(1i32, 2, 3).into_iter().collect();
    let mut b: HashSet<i32> = vec!(2i32, 3, 4).into_iter().collect();

    assert!(a.insert(4));
    assert!(a.contains(&4));

     // `HashSet::insert()` renvoie false si
     // une valeur était déjà présente.
     // assert!(b.insert(4), "Value 4 is already in set B!");
     // FIXME ^ Commentez/décommentez cette ligne

     b.insert(5);

     // Si le type d'un élément de la collection implémente le trait `Debug`,
     // alors la collection devra, elle aussi, implémenter `Debug`.
     // Elle affiche généralement ses éléments dans le format `[elem1, elem2, ...]`.
     println!("A: {:?}", a);
     println!("B: {:?}", b);

     // Affiche [1, 2, 3, 4, 5] dans un ordre arbitraire.
     println!("Union: {:?}", a.union(&b).collect::<Vec<&i32>>());

     // Ceci devrait afficher [1].
     println!("Difference: {:?}", a.difference(&b).collect::<Vec<&i32>>());

     // Affiche [2, 3, 4] dans un ordre arbitraire.
     println!("Intersection: {:?}", a.intersection(&b).collect::<Vec<&i32>>());

     // Affiche [1, 5].
     println!("Symmetric Difference: {:?}",
              a.symmetric_difference(&b).collect::<Vec<&i32>>());
}
```

Les exemples originaux proviennent de la **documentation**.

18 - Outils standards

Bien d'autres types sont fournis par la bibliothèque standard pour supporter des choses telles que :

- Les fils d'exécution ;
- Les canaux ;
- Les opérations sur le système de fichiers.

Ces types vont bien au-de-là de ce que **les primitifs** fournissent.

Voir aussi

Les primitifs et **la bibliothèque standard**.

18-1 - Les fils d'exécution

Rust fournit un méchanisme de création de fils d'exécution natifs via la fonction spawn. L'argument de cette fonction est une closure transférable.

```
use std::thread;

static NTHREADS: i32 = 10;

// Ceci est le thread `main`.
fn main() {
    // On créé un vecteur pour récupérer tous les threads
    // enfants qui ont été créés.
    let mut children = vec![];

    for i in 0..NTHREADS {
        // On passe à un autre thread.
        children.push(thread::spawn(move || {
            println!("this is thread number {}", i)
        }));
    }

    for child in children {
        // On attend que le thread se termine. Renvoie un résultat.
        let _ = child.join();
    }
}
```

Ces threads seront programmés par le système d'exploitation.

18-2 - Les canaux

Rust fournit les canaux asynchrones pour la communication entre les threads. Les canaux permettent l'envoi d'un flux unidirectionnel d'information entre deux extrémités : l'envoyeur (Sender) et le receveur (Receiver).

```
use std::sync::mpsc::{Sender, Receiver};
use std::sync::mpsc;
use std::thread;

static NTHREADS: i32 = 3;

fn main() {

    // Les canaux possèdent deux extrémités: l'envoyeur (`Sender<T>`) et le receveur (`Receiver<T>`),
    // où `T` est le type du message à envoyer.
    // (Le typage est optionnel)
    let (tx, rx): (Sender<i32>, Receiver<i32>) = mpsc::channel();

    for id in 0..NTHREADS {
        // L'envoyeur peut être copié.
        let thread_tx = tx.clone();

        // Chaque thread va envoyer son identifiant par le biais du canal.
        thread::spawn(move || {
            // Le thread prend l'ownership sur `thread_tx`.
            // Chaque thread va ajouter un message dans le file d'attente
            // dans le canal.
            thread_tx.send(id).unwrap();
```

```

            // L'envoi est une opération non-bloquante, le thread continuera à
            // s'exécuter après l'envoi de son message.
            println!("thread {} finished", id);
        });
    }

    // Ici, on récupère tous les messages.
    let mut ids = Vec::with_capacity(NTHREADS as usize);
    for _ in 0..NTHREADS {
        // La méthode `recv` choisit une message se trouvant dans le canal et
        // gêlera le thread courant s'il n'y a aucun message.
        ids.push(rx.recv());
    }

    // Montre l'ordre dans lequel les messages ont été envoyés.
    println!("{:?}", ids);
}
```

18-3 - La structure Path

La structure Path représente les chemins de fichiers dans le système de fichiers sous-jacent. Il y a deux variantes de Path :

1 posix::Path, pour les systèmes UNIX-like ;
2 windows::Path, pour Windows.

Le prélude exporte la variante de Path adaptée à la plateforme.

Une instance de Path peut être créée à partir du type OsStr et fournit de nombreuses méthodes pour obtenir des informations à propos du fichier/répertoire sur lequel le chemin pointe.

Notez que, en interne, un objet Path n'est pas représenté par une chaîne de caractères UTF-8 mais est stocké dans un vecteur d'octets (Vec<u8>). En conséquence, la conversion d'un objet Path en &str n'est pas gratuite et peut échouer (un objet Option est renvoyé).

```
use std::path::{Path, PathBuf};
use std::ffi;

fn main() {
    // Création d'un objet `Path` à partir d'une `&'static str`.
    let path = Path::new(".");

    // La méthode `display` renvoie une structure présentable.
    let display = path.display();

     // La méthode `join()` fusionne un chemin avec toutes les ressources
     // possédant une implémentation de la méthode `as_ref()` pour le type `Path` et
     // renvoie un nouveau chemin avec le séparateur spécifique à l'OS.
     let new_path = path.join("a").join("b");

     // Convertit le chemin en une vue sur une string.
     match new_path.to_str() {
         None => panic!("new path is not a valid UTF-8 sequence"),
         Some(s) => println!("new path is {}", s),
     }
}
```

N'hésitez pas à cosnulter les autres méthodes de Path et la structure Metadata.

Voir aussi

OsStr et **Metadata**.

18-4 - La structure File

La structure File représente un fichier qui a été ouvert (contient un descripteur de fichier), et donne les accès lecture et/ou écriture sur le fichier sous-jacent.

Puisque de nombreuses choses peuvent mal se passer lorsqu'une opération est effectuée, toutes les méthodes de File renvoie le type io::Result<T>, lequel étant un alias pour Result<T, io::Error>.

Ceci couvre les potentielles erreurs de toutes les opérations d'entrée/sortie explicites. Grâce à cela, le programmeur peut visualiser toutes les erreurs possibles et est encouragé à les anticiper.

18-4-1 - La méthode open

La méthode statique open() peut être utilisé pour ouvrir un fichier en lecture seule.

Un objet File est responsable d'une ressource, du descripteur de fichier et prend soin de fermer le fichier lorsqu'il est libéré.

```
// Dans le fichier open.rs
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;
use std::path::Path;

fn main() {
    // Crée un chemin vers le fichier désiré.
    let path = Path::new("hello.txt");
     let display = path.display();

     // Ouvre le chemin en lecture seule, renvoie un objet `io::Result<File>`.
     let mut file = match File::open(&path) {
         // La méthode `description` de `io::Error` renvoie une chaîne de caractères
         // qui décrit l'erreur.
         Err(why) => panic!("couldn't open {}: {}", display,
                                                    why.description()),
         Ok(file) => file,
     };


 // Lit le contenu du fichier dans une chaîne de caractères, renvoie un objet `io::Result<usize>`.
     let mut s = String::new();
     match file.read_to_string(&mut s) {
         Err(why) => panic!("couldn't read {}: {}", display,
                                                    why.description()),
         Ok(_) => print!("{} contains:\n{}", display, s),
     }

     // `file` sort du contexte, le flux ouvert sur le fichier "hello.txt"
     // va être fermé.
}
```

Voici le résultat attendu :

```
$ echo "Hello World!" > hello.txt
$ rustc open.rs && ./open
hello.txt contains:
Hello World!
```

Nous vous encourageons à confronter l'exemple précédent à des cas d'échec différents (e.g. hello.txt n'existe pas, hello.txt ne peut pas être lu).

18-4-2 - La méthode create

La méthode statique create() ouvre un fichier en écriture seule. Si le fichier existe déjà, le contenu sera écrasé. Autrement, un nouveau fichier sera créé.

```
// Dans le fichier create.rs
static LOREM_IPSUM: &'static str =
"Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
";

use std::error::Error;
use std::io::prelude::*;
use std::fs::File;
use std::path::Path;

fn main() {
    let path = Path::new("out/lorem_ipsum.txt");
    let display = path.display();

    // Ouvre un fichier en écriture seule, renvoie un objet `io::Result<File>`.
    let mut file = match File::create(&path) {
        Err(why) => panic!("couldn't create {}: {}",
                           display,
                           why.description()),
        Ok(file) => file,
    };

    // Écrit la chaîne de caractères de `LOREM_IPSUM` dans `file`, renvoie
    // un objet `io::Result<()>`.
    match file.write_all(LOREM_IPSUM.as_bytes()) {
        Err(why) => {
            panic!("couldn't write to {}: {}", display,
                                               why.description())
        },
        Ok(_) => println!("successfully wrote to {}", display),
    }
}
```

Voici le résultat attendu :

```
$ mkdir out
$ rustc create.rs && ./create
successfully wrote to out/lorem_ipsum.txt
$ cat out/lorem_ipsum.txt
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse
cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
```

Comme pour l'exemple précédent, nous vous encourageons à confronter l'exemple à d'autres cas où l'opération pourrait échouer.

Il existe également d'autres outils pouvant ouvrir des fichiers dans des modes différents tels que : read+write, append, etc.

18-5 - Les sous-processus

La structure process::Output représente la sortie d'un sous-processus terminé et la structure process::Command est un constructeur de processus.

```
use std::process::Command;

fn main() {
    let output = Command::new("rustc")
        .arg("--version")
        .output().unwrap_or_else(|e| {
            panic!("failed to execute process: {}", e)
    });

    if output.status.success() {
        let s = String::from_utf8_lossy(&output.stdout);

        print!("rustc succeeded and stdout was:\n{}", s);
    } else {
        let s = String::from_utf8_lossy(&output.stderr);

        print!("rustc failed and stderr was:\n{}", s);
    }
}
```

Nous vous encourageons à essayer l'exemple précédent en lançant rustc avec un flag incorrect.

18-5-1 - Les pipes

La structure Process représente un processus en cours d'exécution et expose la gestion de stdin, stdout et stderr pour interagir avec le processus sous-jacent via les pipes.

```
use std::error::Error;
use std::io::prelude::*;
use std::process::{Command, Stdio};

static PANGRAM: &'static str =
"the quick brown fox jumped over the lazy dog\n";

fn main() {
    // On crée un processus dans lequel la commande `wc` va s'exécuter.
    let process = match Command::new("wc")
                                .stdin(Stdio::piped())
                                .stdout(Stdio::piped())
                                .spawn() {
        Err(why) => panic!("couldn't spawn wc: {}", why.description()),
        Ok(process) => process,
    };

    // On écrit quelque chose dans l'entrée standard de `wc`.
    // `stdin` est de type `Option<ChildStdin>`, mais puisque nous savons que
    // cette instance en possède une, nous pouvons directement l'`unwrap()`.
    match process.stdin.unwrap().write_all(PANGRAM.as_bytes()) {
        Err(why) => panic!("couldn't write to wc stdin: {}",
                           why.description()),
        Ok(_) => println!("sent pangram to wc"),
    }

    // Puisque `stdin` ne survit pas après l'appel du dessus, elle va être libérée et
    // et le pipe fermé.
    //
    // C'est très important sinon `wc` ne pourrait pas commencer à traiter l'entrée
    // que nous avons soumis.


  // La champ `stdout` est également de type `Option<ChildStdout>` et doit donc être `unwrap()`.
```

```
    let mut s = String::new();
    match process.stdout.unwrap().read_to_string(&mut s) {
        Err(why) => panic!("couldn't read wc stdout: {}",
                           why.description()),
        Ok(_) => print!("wc responded with:\n{}", s),
    }
}
```

18-5-2 - La méthode wait

Vous souhaiteriez peut-être attendre qu'un processus, dont un objet process::Child est responsable, se termine. Pour cela vous devez appeler la méthode Child::wait qui renverra un objet process::ExitStatus.

```
// Dans le fichier wait.rs
use std::process::Command;

fn main() {
    let mut child = Command::new("sleep").arg("5").spawn().unwrap();
    let _result = child.wait().unwrap();

    println!("reached end of main");
}
```

```
$ rustc wait.rs && ./wait
reached end of main
# `wait` s'est exécuté pendant 5 secondes.
# Une fois la commande `sleep 5` terminée notre programme `wait` a pris fin.
```

18-6 - Opérations sur le système de fichiers

Le module std::io::fs contient de nombreuses fonctions traitant avec le système de fichiers.

```
// Dans le fichier fs.rs
use std::fs;
use std::fs::{File, OpenOptions};
use std::io;
use std::io::prelude::*;
use std::os::unix;
use std::path::Path;

// Une simple implémentation de la commande `cat path`.
fn cat(path: &Path) -> io::Result<String> {
    let mut f = try!(File::open(path));
    let mut s = String::new();
    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}

// Une simple implémentation de la commande `echo s > path`.
fn echo(s: &str, path: &Path) -> io::Result<()> {
    let mut f = try!(File::create(path));

    f.write_all(s.as_bytes())
}

// Une simple implémentation de la commande `touch path` (ignore les fichiers existants).
fn touch(path: &Path) -> io::Result<()> {
    match OpenOptions::new().create(true).write(true).open(path) {
        Ok(_) => Ok(()),
        Err(e) => Err(e),
    }
}

fn main() {
```

```
    println!("`mkdir a`");
    // Crée un répertoire et renvoie un objet `io::Result<()>`.
    match fs::create_dir("a") {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(_) => {},
    }

    println!("`echo hello > a/b.txt`");
    // Le match précédent peut être simplifié en utilisant la méthode `unwrap_or_else()`.
    echo("hello", &Path::new("a/b.txt")).unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`mkdir -p a/c/d`");
    // Crée un répertoire récursivement, renvoie un objet `io::Result<()>`.
    fs::create_dir_all("a/c/d").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`touch a/c/e.txt`");
    touch(&Path::new("a/c/e.txt")).unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`ln -s ../b.txt a/c/b.txt`");
    // Crée un lien symbolique, renvoie un objet `io::Result<()>`.
    if cfg!(target_family = "unix") {
        unix::fs::symlink("../b.txt", "a/c/b.txt").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
        });
    }

    println!("`cat a/c/b.txt`");
    match cat(&Path::new("a/c/b.txt")) {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(s) => println!("> {}", s),
    }

    println!("`ls a`");
    // Lit le contenu d'un répertoire, renvoie un objet `io::Result<Vec<Path>>`.
    match fs::read_dir("a") {
        Err(why) => println!("! {:?}", why.kind()),
        Ok(paths) => for path in paths {
            println!("> {:?}", path.unwrap().path());
        },
    }

    println!("`rm a/c/e.txt`");
    // Supprime un fichier, renvoie un objet `io::Result<()>`.
    fs::remove_file("a/c/e.txt").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });

    println!("`rmdir a/c/d`");
    // Supprime un répertoire vide, renvoie un objet `io::Result<()>`.
    fs::remove_dir("a/c/d").unwrap_or_else(|why| {
        println!("! {:?}", why.kind());
    });
}
```

Voici le résultat attendu :

```
$ rustc fs.rs && ./fs
`mkdir a`
`echo hello > a/b.txt`
`mkdir -p a/c/d`
`touch a/c/e.txt`
`ln -s ../b.txt a/c/b.txt`
`cat a/c/b.txt`
> hello
`ls a`
```

```
> "a/b.txt"
> "a/c"
`rm a/c/e.txt`
`rmdir a/c/d`
```

Et l'état final du répertoire a est :

```
$ tree a
a
|-- b.txt
`-- c
    `-- b.txt -> ../b.txt

1 directory, 2 files
```

Voir aussi

La macro cfg!.

18-7 - Les arguments du programme

Les arguments passés en ligne de commande peuvent être récupérés en utilisant std::env::args qui renvoie un itérateur fournissant une String pour chaque argument :

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    // Le premier argument est le chemin à partir duquel le programme
    // a été appelé.
    println!("My path is {}.", args[0]);

     // Le reste des arguments sont ceux passés en ligne de commande au programme.
     // On appelle le programme comme ceci:
     //  $ ./args arg1 arg2
     println!("I got {:?} arguments: {:?}.", args.len() - 1, &args[1..]);
}
```

```
$ ./args 1 2 3
My path is ./args.
I got 3 arguments: ["1", "2", "3"].
```

18-7-1 - Récupération des arguments

Le pattern matching peut être utilisé pour traiter de simples arguments :

```
use std::env;

fn increase(number: i32) {
    println!("{}", number + 1);
}

fn decrease(number: i32) {
    println!("{}", number - 1);
}

fn help() {
     println!("usage:
match_args <string>
     Check whether given string is the answer.
match_args {{increase|decrease}} <integer>
     Increase or decrease given integer by one.");
```

```rust
}

fn main() {
    let args: Vec<String> = env::args().collect();

    match args.len() {
        // Pas d'arguments passés.
        1 => {
            println!("My name is 'match_args'. Try passing some arguments!");
        },
        // Un seul argument passé.
        2 => {
            match args[1].parse() {
                Ok(42) => println!("This is the answer!"),
                _ => println!("This is not the answer."),
            }
        },
        // Une commande et un argument passé.
        3 => {
            let cmd = &args[1];
            let num = &args[2];
            // On traite le nombre.
            let number: i32 = match num.parse() {
                Ok(n) => {
                    n
                },
                Err(_) => {
                    println!("error: second argument not an integer");
                    help();
                    return;
                },
            };
            // On traite la commande.
            match &cmd[..] {
                "increase" => increase(number),
                "decrease" => decrease(number),
                _ => {
                    println!("error: invalid command");
                    help();
                },
            }
        },
        // On couvre tous les autres cas...
        _ => {
            // ... en affichant l'aide.
            help();
        }
    }
}
```

```
$ ./match_args Rust
This is not the answer.
$ ./match_args 42
This is the answer!
$ ./match_args do something
error: second argument not an integer
usage:
match_args <string>
    Check whether given string is the answer.
match_args {increase|decrease} <integer>
    Increase or decrease given integer by one.
$ ./match_args do 42
error: invalid command
usage:
match_args <string>
    Check whether given string is the answer.
match_args {increase|decrease} <integer>
    Increase or decrease given integer by one.
$ ./match_args increase 42
43
```

18-8 - FFI

Rust fournit une Interface pour Fonction Externe (« Foreign Function Interface », dans la langue de Shakespear) pour les bibliothèques écrites en C. Les fonctions externes peuvent être déclarées dans un bloc extern annoté de l'attribut #[link] contenant le nom de la bibliothèque externe.

```
// Dans le fichier ffi.rs
use std::fmt;

// Ce bloc externe lie la bibliothèque libm.
#[link(name = "m")]
extern {
    // Ceci est une fonction externe
    // qui calcule la racine carrée d'un nombre complexe à précision simple.
    fn csqrtf(z: Complex) -> Complex;
}

fn main() {
    // z = -1 + 0i
    let z = Complex { re: -1., im: 0. };

    // Appeler une fonction externe est une opération dite "à risque".
    let z_sqrt = unsafe {
        csqrtf(z)
    };

    println!("the square root of {:?} is {:?}", z, z_sqrt);
}

// Implémentation minimale d'un nombre complex à précision simple.
#[repr(C)]
#[derive(Clone, Copy)]
struct Complex {
    re: f32,
    im: f32,
}

impl fmt::Debug for Complex {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        if self.im < 0. {
            write!(f, "{}-{}i", self.re, -self.im)
        } else {
            write!(f, "{}+{}i", self.re, self.im)
        }
    }
}
```

```
$ rustc ffi.rs && ./ffi
the square root of -1+0i is 0+1i
```

Puisque l'appel de fonctions externes est considéré comme « à risque », il est courant d'écrire des wrappers sécurisés.

```
// Dans le fichier safe.rs
use std::fmt;

#[link(name = "m")]
extern {
    fn ccosf(z: Complex) -> Complex;
}

// Wrapper sécurisé.
fn cos(z: Complex) -> Complex {
    unsafe { ccosf(z) }
}

fn main() {
    // z = 0 + 1i
    let z = Complex { re: 0., im: 1. };
```

```
    println!("cos({:?}) = {:?}", z, cos(z));
}

// Implémentation minimale d'un nombre complexe à précision simple.
#[repr(C)]
#[derive(Clone, Copy)]
struct Complex {
    re: f32,
    im: f32,
}

impl fmt::Debug for Complex {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        if self.im < 0. {
            write!(f, "{}-{}i", self.re, -self.im)
        } else {
            write!(f, "{}+{}i", self.re, self.im)
        }
    }
}
```

19 - Divers

Certaines sections ne sont pas tout à fait liées à la manière dont vous programmez mais vous introduisent et fournissent des outils facilitant les choses pour tout le monde. Ces sections abordent :

- La documentation : Générez la documentation d'une bibliothèque pour les utilisateurs via `rustdoc` ;
- Les tests : Créez des suites de tests pour garantir l'intégrité de vos bibliothèques ;
- Les benchmarks : Créez des benchmarks de vos fonctionnalités pour garantir leur performance.

19-1 - Documentation

Les commentaires de documentation sont très utiles pour d'importants projets nécessitant une documentation. Lorsque vous lancez **Rustdoc**, ce sont ces commentaires qui seront compilés dans la documentation. Ils sont préfixés par la séquence `///` et supportent **Markdown**.

```
// #![crate_name = "doc"]

/// Un être humain est représenté ici.
pub struct Person {
    /// Une personne doit avoir un nom.
    name: String,
}

impl Person {
    /// Renvoie une personne avec le nom qu'on lui a donné.
    ///
    /// # Arguments
    ///
    /// * `name` - Une slice qui contient le nom de la personne.
    ///
    /// # Example
    ///
    /// ```
    /// // Vous pouvez écrire du code rust entre les balises
    /// // dans les commentaires.
    /// // Si vous passez --test à Rustdoc, il testera même la source pour vous !
    /// use doc::Person;
    /// let person = Person::new("name");
    /// ```
    pub fn new(name: &str) -> Person {
        Person { name: name.to_string() }
    }

    /// Affiche un salut amical !
```

```
    ///
    /// Affiche "Hello, [name]" à l'objet `Person` en question.
    pub fn hello(&self) {
        println!("Hello, {}!", self.name);
    }
}

fn main() {
    let john = Person::new("John");

    john.hello();
}
```



***Note** : si vous souhaitez compiler ce programme vous-même, n'oubliez pas de décommenter la ligne // #![crate_name = "doc"].*

Pour lancer les tests, vous devez tout d'abord compiler le code en mode bibliothèque puis renseigner la position de la bibliothèque à rustdoc pour qu'il puisse la lier à chaque programme de la documentation :

```
rustc doc.rs --crate-type lib
rustdoc --test --extern doc="libdoc.rs"
```

Lorsque vous exécutez la commande cargo test sur une crate, Cargo générera et exécutera automatiquement les commandes respectives de rustc et rustdoc.

19-2 - Tests

Les fonctions peuvent être testées en utilisant ces **attributs**:

- #[test] désigne une fonction comme test unitaire. La fonction ne doit prendre aucun paramètre et ne rien renvoyer ;
- #[should_panic] désigne une fonction comme un test voué à l'échec.

```
// Dans le fichier unit_test.rs
// Compile `main` à condition que la compilation des tests ne soit pas activée.
#[cfg(not(test))]
fn main() {
    println!("If you see this, the tests were not compiled nor ran!");
}

// Compile le module `test` seulement si la compilation des tests est activée.
#[cfg(test)]
mod test {
// Un test unitaire `distance_test` est nécessaire.
    fn distance(a: (f32, f32), b: (f32, f32)) -> f32 {
        (
            (b.0 - a.0).powi(2) +
            (b.1 - a.1).powi(2)
        ).sqrt()
    }

    #[test]
    fn distance_test() {
        assert!(distance((0f32, 0f32), (1f32, 1f32)) == (2f32).sqrt());
    }

    #[test]
    #[should_panic]
    fn failing_test() {
        assert!(1i32 == 2i32);
    }
}
```

Les tests peuvent être exécutés avec la commande `cargo test` ou `rustc --test`.

```
$ rustc --test unit_test.rs
$ ./unit_test

running 2 tests
test test::distance_test ... ok
test test::failing_test ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Si `--test` n'a pas été inclut alors il devrait se passer ceci :

```
$ rustc unit_test.rs
$ ./unit_test
If you see this, the tests were not compiled nor ran!
```

Voir aussi

Les attributs, **la compilation conditionnelle** et **mod**.

20 - Opérations à risque

Pour reprendre ce que la documentation officielle dit : « Il faudrait essayer de minimiser la quantité de code à risque dans la base du code. » Avec ceci en tête, commençons ! Les blocs unsafe en Rust sont utilisés pour contourner les protections mises en place par le compilateur ; plus précisément, il y a quatre principaux cas d'utilisation que nous pouvons retrouver dans les blocs unsafe :

1 Le déréférencement des pointeurs bruts ;
2 L'appel d'une fonction à partir de la FFI (cette partie est couverte à d'autres endroits dans le livre) ;
3 La modification des types par le biais de `std::mem::transmute` ;
4 Inliner de l'assembleur.

Les pointeurs bruts

Les pointeurs bruts `*` et références `&T` fonctionnent de la même manière mais les références sont toujours sécurisées parce qu'elles garantissent de pointer sur une ressource valide grâce au vérificateur d'emprunts. Le déréférencement d'un pointeur brut ne peut se faire que par le biais d'un bloc unsafe.

```rust
// Dans le fichier pointer.rs
fn main() {
    let raw_p: *const u32 = &10;

    unsafe {
        assert!(*raw_p == 10);
    }
}
```

Transmuter

Permet une simple conversion d'un type à l'autre, cependant les deux types doivent disposer de la même taille en mémoire et le même alignement :

```rust
// Dans le fichier transmute.rs
fn main() {
     let u: &[u8] = &[49, 50, 51];

     unsafe {
         assert!(u == std::mem::transmute::<&str, &[u8]>("123"));
     }
```

```
}
```

1 : Consultez **cette section** pour plus de détails.